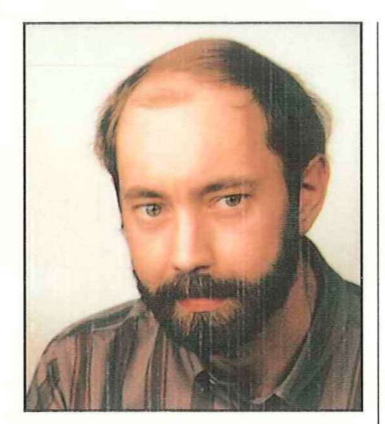
Markt&Technik

Gerd Möllmann

C128

PROGRAMMIEREN IN MASCHINENSPRACHE

Ein Lehr- und Übungsbuch mit ausgewählten ROM- und RAM-Routinen für die Entwicklung von eigenen Assemblerprogrammen.



C 128 PROGRAMMIEREN IN MASCHINENSPRACHE

GERD BERNHARD MÖLLMANN, geboren am 3. 1. 1957 in Borken/Westfalen, legte 1975 das Abitur am Gymnasium seiner Heimatstadt ab. Das anschließende Studium der Mathematik, mit Schwerpunkt auf dem numerischen Zweig, führte zwangsläufig zu ersten Programmieraufgaben - zunächst auf Großrechnern in verschiedenen Hochsprachen. Im Laufe des unaufhaltsamen Siegeszuges der Mikroprozessoren während dieser Zeit erwarb der Autor auch die ersten Erfahrungen mit ihrer Programmierung. Da zu diesem Zeitpunkt an preisgünstige Computer zum persönlichen Gebrauch noch nicht zu denken war, gehört der Autor noch zu denjenigen, denen das zweifelhafte Vergnügen zuteil wurde, Maschinensprache Bit für Bit am LED statt am Bildschirm lernen zu müssen. 1985 fand dann der Schritt in eine freiberufliche Tätigkeit statt. Seither sind verschiedene Softwareentwicklungen und Veröffentlichungen entstanden, weitere werden folgen.

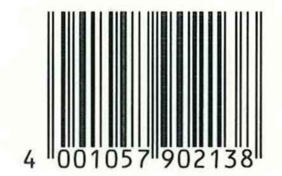
Wer die Fähigkeiten des Commodore 128 über das BASIC-Betriebssystem hinaus voll ausschöpfen will, der muß, wie bereits bei seinem Vorgänger, dem so erfolgreichen C 64, in Maschinensprache programmieren. Von BASIC zu Maschinensprache - dieser nicht immer reibungslose Umstieg steht im Vordergrund des Buches.
Besonderer Wert wird auf die Verwendung von ROM- und RAM-Routinen aus BASIC und Betriebssystem gelegt. Auf diese Weise entstehen erste systemnahe Programme. Zum grundlegenden Verständnis der Hardware des Commodore 128 ist eine ausführliche Beschreibung der in diesem Computer verwendeten Bausteingruppen enthalten.

Aus dem Inhalt:

- Registerbeschreibung des VIC, VDC, SID, der CIA
- Die Speicherverwaltung des C 128 (Memory Management Unit)
- BASIC-Routinen in der Common Area
- Kernal-Routinen in der Common Area
- Kernal-Vektoren in der erweiterten Zeropage und ihre Nutzung
- BASIC-Vektoren
- Usertoken und ihre Anwendung
- Häufig benutzte BASIC-Routinen
- FRMEVL
- Häufig benutzte Kernal-Routinen

Hardwareanforderung: Commodore 128 oder Commodore 128 D

ISBN 3-89090-213-8



Markt&Technik

DM 52,-
sFr. 47,80
öS 405,60



Gerd Möllmann

C128 Programmieren in Maschinensprache

Ein Lehr- und Übungsbuch mit ausgewählten ROM- und RAM-Routinen für die Entwicklung von eigenen Assemblerprogrammen

Markt&Technik Verlag

CIP-Kurztitelaufnahme der Deutschen Bibliothek

Möllmann, Gerd:
C 128 programmieren in Maschinensprache : e. Lehr- u. Übungsbuch mit ausgewählten ROM- u. RAM-Routinen für d. Entwicklung von eigenen Assemblerprogrammen/ Gerd Möllmann. – Haar bei München : Markt-und-Technik-Verlag, 1986.
ISBN 3-89090-213-8

15 14 13 12 11 10 9 8 7 6 5 4 3
89 88 87

ISBN 3-89090-213-8

Einbandgestaltung: Grafikdesign Heinz Rauner
Druck: Schoder, Gersthofen
Printed in Germany

Inhaltsverzeichnis

Vorwort

Das vorliegende Buch ist nicht als Lehrbuch für die Assemblerprogrammierung als solche gedacht. Es soll vielmehr dem mit der 6502-Maschinensprache Vertrauten den Weg in das Innenleben des C128 ebnen.
Natürlich kann man nicht erwarten, durch den Erwerb eines Buches automatisch zum erfolgreichen Assemblerprogrammierer zu werden. Dazu bedarf es in der Regel wohl eines jahrelangen Sammelns von Erfahrungen und damit des Erwerbs eines Schatzes von Tricks und Kniffen, der von den meisten Programmierern auch als solcher gehütet wird. Ich hoffe allerdings, daß im vorliegenden Buch eine ganze Reihe solcher Tricks enthalten sind, jedenfalls, was den speziellen Umgang mit dem 128er anbelangt.
Das Buch ist von seinem Aufbau her nicht dazu vorgesehen, an einem Stück als Lehrbuch gelesen zu werden. Es war vielmehr meine Absicht, sachbezogene Informationen zusammenzufassen, so daß sie dann, wenn sie gebraucht werden, zur Verfügung stehen. Dabei wurde Wert darauf gelegt, daß zu wichtigen Themen jeweils auch ein Beispielprogramm gebracht ist, soweit dies möglich war.

Einleitung

Dieses Buch ist streng in Kapitel unterteilt, die zu unterschiedlichen Sachbereichen Bezug nehmen.
Das erste Kapitel gibt eine Zusammenfassung der Hardware des C128. Dazu gehören die Verwaltung der Speicherbänke durch die Memory Management Unit, die beiden Videocontroller, die Ein- und Ausgabebausteine sowie der Soundchip SID. Dabei wurde besonderer Wert auf die Beschreibung des 80-Zeichen-Videocontrollers sowie der MMU gelegt, da diese Bausteine gegenüber dem C64 neu sind, und keine Beschreibung dieser Chips in der schon weit verbreiteten C64-Lektüre zu finden ist, wie es bei den anderen Chips der Fall ist, die auch schon im C64 eingebaut waren.
Das zweite Kapitel befaßt sich mit der eingehenden Erläuterung einiger Routinen in der sogenannten Common Area, deren genaue Kenntnis für den Assemblerprogrammierer durch das interne Banking im 128er unerläßlich ist.
Das dritte Kapitel ist der Beschreibung des 128er-Betriebssystems vorbehalten. Die Tastatur, Ein- und Ausgabe vom beziehungsweise auf den Bildschirm, die Arbeitsweise des Kernal-Editors sowie die Behandlung des Commodore-spezifischen IEC-Bus, speziell der Umgang mit einer angeschlossenen Floppy, bilden die Hauptpunkte dieses Kapitels. Hinzu kommt eine eingehende Beschreibung der wichtigen Kernal-Routinen und der Umgang mit diesen.
In Anbetracht dessen, daß der Datenrecorder sich beim 128er wohl kaum durchsetzen wird, da dieser die Fähigkeiten des C128 in keiner Weise unterstützt, wurde auf die entsprechenden Routinen des Betriebssystems hierzu nicht eingegangen. Ähnlich verhält es sich mit den RS232-Routinen. Ihre Beschreibung soll speziellen Büchern vorbehalten bleiben, die etwa das Thema Daten-Fern-Übertragung behandeln, da zu ihrer Benutzung Zusatzgeräte am Rechner erforderlich sind.
Im Kapitel vier wird die Trickkiste ein gutes Stück geöffnet. Dieses Kapitel ist den Kernal-Vektoren gewidmet, die erhebliche Eingriffe in das Betriebssystem gestatten. An Beispielprogrammen wird gezeigt, welche interessanten Anwendungen möglich sind.

Das fünfte Kapitel schließlich geht auf den BASIC-Interpreter aus der Sicht des Maschinenprogrammierers ein. Wie ist der Programmtext aufgebaut, wie rechnet der Interpreter, wie behandelt er Variable, sind nur einige der Themen, die an dieser Stelle behandelt werden. Der praktische Umgang mit den gewonnenen Erkenntnissen steht dabei absolut im Vordergrund, wie die vielen Beispielprogramme belegen. Die wichtigen Routinen des Interpreters sind eingehend erläutert und aufgeführt. Grafik, Sound und Sprites wurden allerdings weggelassen, da diese Themen zum einen in schon vorhandenen Büchern zum C64 zu finden sind, zum zweiten diese drei Bereiche von BASIC so gut unterstützt werden, daß sie für das Programmieren in Maschinensprache weniger interessant geworden sind.
Kapitel 6: Auch das BASIC 7.0 kann durch eigene Routinen sinnvoll ergänzt werden. Ermöglicht wird dies wie beim Betriebssystem durch Vektoren. An vielen Beispielen wird gezeigt, wie es gemacht wird, eine kleine komplette Befehlserweiterung gehört dazu.
Die Zeropage und ihre nähere Umgebung sind Thema des siebten Kapitels, das zugleich auch das letzte dieses Buches ist. Diese geheimnisvollen toten Briefkästen der Assemblerprogrammierer, wie einmal über die Zeropage geschrieben wurde, entpuppen sich bei näherer Betrachtung und Kenntnis der vorangegangenen Kapitel als eine Ansammlung äußerst interessanter und nützlicher Speicherzellen, deren Kenntnis eine der Voraussetzungen für eine erfolgreiche Programmierarbeit ist.
Alle enthaltenen Beispielprogramme sind ausgetestet und funktionstüchtig, sofern das Druckfehlerteufelchen nicht doch zuschlagen sollte. In ihnen sind einige Tricks enthalten, von denen ich hoffe, eigentlich sogar sicher bin, daß Sie Ihnen bei der weiteren Beschäftigung mit dem 128er noch gute Dienste leisten werden.

1 Inside Commodore 128

Thema: Speicherverwaltung des Commodore 128, 80-Zeichen-Videocontroller, Kurzbeschreibungen des Soundchips, der CIAs und des 40-Zeichen-Controllers.

Bei näherem Bekanntwerden mit dem Commodore 128 kann man feststellen, daß einige Chips, darunter Soundchip, VIC II und die CIAs, praktisch unverändert vom C64 übernommen wurden. Eine Tatsache, die eigentlich nicht weiter verwunderlich ist, wenn man bedenkt, daß der C128 einen Modus enthält, der den C64 weitgehend simuliert. Auf jeden Fall dürfen sich diejenigen freuen, die schon mit dem C64 gearbeitet haben, denn die Programmierung dieser Chips wird ihnen vertraut sein – sogar die Adressen der einzelnen Register dieser Bausteine sind mit denen beim C64 identisch.
Sicher kann man innerhalb eines Buches nicht sämtliche Möglichkeiten aufzeigen, die in den Bausteinen des Commodore 128 stecken, dieses Kapitel soll deshalb nur ein Basiswissen vermitteln. Da schon etliche Bücher über die Grafikprogrammierung des VIC II und die Programmierung des SID erschienen sind, kann ich bei weitergehenden Fragen mit gutem Gewissen auf diese verweisen.

1.1 Die Speicherverwaltung des 128 (die MMU)

Auf den ersten Blick ist es schon erstaunlich: Ein 8-Bit-Prozessor, der von Haus aus nur einen Adreßraum von 64 KByte verwalten kann, bedient im 128er einen Speicher, der bis zu 256 KByte und mehr ausgebaut werden kann. Und es stellt sich natürlich die Frage, wie dies erreicht wird.
Das Stichwort hierzu heißt Banking und wird den Programmierer in Maschinensprache wohl die erste Zeit noch im Schlaf verfolgen. Ohne Banking läuft im C128 rein gar nichts,

man tut deshalb gut daran, sich eingehend mit dieser Art der Speicherverwaltung zu befassen.
Verantwortlich für das Banking zeichnet die Memory Management Unit, kurz MMU genannt. Sie übernimmt in der Speicherverwaltung des 128ers den Part, den der Prozessor selbst nicht leisten kann. Die MMU, oder besser gesagt die Einstellungen, die der Programmierer an der MMU vornimmt, bestimmen, welchen Teil des Gesamtspeichers des C128 der Prozessor innerhalb seines 64K-Adreßraums zu Gesicht bekommt. Alle nicht eingeblendeten Teile des Speichers kann der Prozessor nicht wahrnehmen. Wenn man den gesamten Speicher des C128 nutzen will, wird man, der Not und nicht dem Triebe gehorchend, also ständig an der MMU basteln müssen. Wer schon mit dem C64 gearbeitet hat, kennt eine ähnliche Einrichtung wie die MMU – die Speicherstelle 1 in der 64er-Zeropage. Auch mit ihr kann man durch Ein- und Ausblenden von ROMs mehr als 64 KByte verwalten.
Sehen wir uns einmal an, welche Möglichkeiten uns die insgesamt 11 Register der MMU bieten:

1.1.1 Das Konfigurationsregister (CR, $d500 $ff00)

Das sogenannte Konfigurationsregister ist das Register der MMU, das in seiner Wirkung der Speicherstelle 1 im C64 am nächsten kommt. Im Speicher ist es an der Adresse $d500 im In/Out-Bereich zu finden. Da dieser Bereich abgeschaltet werden muß, um an das parallel liegende Zeichensatz-ROM zu kommen, liegt eine Kopie des Registers an der Adresse $ff00. An dieser Stelle befindet sich ein kleines Fenster in allen ROMs und RAMs, so daß das Konfigurationsregister bei $ff00 stets sichtbar ist. Will man andere Register der MMU ändern als das Konfigurationsregister, muß man gegebenenfalls erst durch Änderung des Konfigurationsregisters bei $ff00 dafür sorgen, daß der I/O-Bereich eingeschaltet wird.
Was läßt sich im Konfigurationsregister ein- und ausschalten?
Zunächst läßt sich ein Teilbereich des insgesamt verfügbaren RAM-Speichers auswählen, der sozusagen die Grundlage des für den Prozessor sichtbaren Speichers bildet. Der gesamte Speicher wird dazu in Blöcke zu je 64 KByte unterteilt, die durchnumeriert werden. In der jetzigen Version des C128 sind maximal vier solcher 64K-Speicherbänke möglich, zwei, die Nummern 0 und 1, sind serienmäßig eingebaut. Spätere Versionen sollen evtl. 16 Bänke, also 1 MByte, erlauben. Im Konfigurationsregister sind die beiden höchstwertigen Bits für die Auswahl der RAM-Bank verantwortlich. Es bedeutet

Bit 7	Bit 6	ausgewählte Bank
0	0	0
0	1	1
1	0	2
1	1	3

In der Grundversion des C128 mit nur zwei Bänken hat das Bit 7 keine Bedeutung, so daß beim Anwählen von Bank 2 die Bank 0 angesprochen wird, entsprechend bei der Wahl von Bank 3 die erste Bank.
Als nächstes können in die so ausgewählte RAM-Bank wie beim C64 ROMs eingeblendet werden. Im 128er wurde dabei gleich die Möglichkeit von ROM-Erweiterungen bedacht, die im Computer selbst (interne ROMs), oder außen (externe ROMs) angebracht werden können. Die verschiedenen Bereiche, in die ROMs eingeblendet werden können, sind:

$4000-$7fff

Hier kann der untere Teil des eingebauten BASIC-ROMs eingeschaltet werden. Das zweitniedrigste Bit, Bit 1, ist im Konfigurationsregister dafür zuständig.

Bit 1	
0	BASIC-ROM in diesem Bereich ist an
1	Es ist die mit Bit 7 und 6 angewählte RAM-Bank sichtbar

Ein interessanter Effekt stellt sich ein, wenn man versucht, eine Adresse bei eingeschaltetem ROM zu beschreiben. Der Schreibzugriff führt dann durch das ROM hindurch auf die, wenn man so will, darunterliegende RAM-Bank. Durch die ROMs kann also, bildlich gesprochen, wie schon beim C64 hindurchgePOKEt werden.

$8000-$bfff

In diesen Bereich kann der obere Teil des BASIC-ROMs, der zugleich den eingebauten Monitor enthält, eingeblendet werden. Ferner können externes und internes ROM angewählt werden. Die Auswahl wird durch die Bits 3 und 2 getroffen:

Bit 3	Bit 2	
0	0	Oberes BASIC-ROM ist eingeschaltet
0	1	Internes ROM ist an
1	0	Externes ROM
1	1	Die RAM-Bank ist sichtbar

Sind internes und externes ROM nicht vorhanden, spielt das Bit 3 keine Rolle. Es gilt dann: ROM ist gewählt bei Bit 2 = 0, andernfalls ist RAM sichtbar.

$c000-$cfff und $e000-$ffff

Diese Bereiche werden gemeinsam geschaltet. Im Bereich von $c000 bis $ffff befindet sich also ein Loch, in dem der I/O-Bereich und das Character-ROM liegt. Ein weiteres kleine-

res Fenster liegt von $ff00 bis $ff04, wo das nicht abschaltbare Konfigurationsregister zusammen mit vier weiteren Registern der MMU, die später besprochen werden, untergebracht ist. Für den Kernal-Bereich sind Bit 5 und 4 des Konfigurationsregisters verantwortlich:

Bit 5	Bit 4	
0	0	Das Kernal-ROM ist eingeschaltet
0	1	Internes ROM
1	0	Externes ROM
1	1	RAM-Bank

Auch hier gilt, daß Bit 5 keine Bedeutung hat, solange keine ROM-Erweiterungen eingebaut oder angeschlossen sind.

$d000-$dfff

Als letztes bleibt noch der Bereich, in dem I/O und Zeichensatz-ROM zu finden sind. Dieser Bereich kann von der I/O (VIC-Register, CIAs, MMU etc.), vom Character-ROM oder von RAM eingenommen werden. Die Steuerung erfolgt durch Bit 0 des Konfigurationsregisters in Zusammenarbeit mit den Bits Nummer 5 und 4:

Bit 0	
0	Der I/O-Bereich ist eingeblendet
1	Es ist die RAM-Bank sichtbar, wenn durch die Bits 5 und 4 RAM eingeschaltet ist, andernfalls ist das Character-ROM sichtbar.

1.1.2 Load- und Präkonfigurationsregister

Zur Erleichterung der Arbeit mit der MMU – wir haben ja gesehen, daß wir praktisch ständig das Konfigurationsregister umschalten müssen, wenn wir die Speicherkapazität des C128 nutzen wollen – existieren die sogenannten Präkonfigurationsregister, die eine Art Vorprogrammierung der MMU ermöglichen.
Im Speicher sind die vier Präkonfigurationsregister (pcra bis pcrd) an den Adressen $d501 bis $d504 zu finden. Um Werte in diese Register schreiben zu können, muß also der I/O-Bereich eingeschaltet werden.
Ein Gegenstück zu den Präkonfigurationsregistern befindet sich jeweils im nicht abschaltbaren Bereich von $ff01 bis $ff04. Diese sogenannten Load-Configuration-Register (lcra bis lcrd) ermöglichen die Übertragung der in den PCRs abgespeicherten, voreingestellten Werte in das Konfigurationsregister. Bei einem Schreibzugriff auf das LCRA gelangt, wie durch das Betätigen eines Sensorschalters, der Inhalt des PCRA in das Konfigurationsregister, bei einem Schreiben in das LCRB erhält das CR den Wert des PCRB usw.

Der Schreibzugriff selbst verändert weder den Wert des LCRs noch den des PCRs. Ein Auslesen der LCRs ergibt die Inhalte der zugehörigen PCRs.
Das im 128er eingebaute BASIC macht ausgiebig Gebrauch von den PCRs und LCRs. Dies sollte man vielleicht schon einmal im Hinterkopf behalten, bevor man die Einstellungen der PCRs ändert. Die von BASIC benutzten Voreinstellungen sind:

PCRA ($d501) =	$3f, dies entspricht einer Konfiguration, in der RAM-Bank 0 ohne ROMs ausgewählt ist.
PCRB ($d502) =	$7f, entspricht PCRA mit der RAM-Bank 1.
PCRC ($d503) =	$01, unter dieser Konfiguration sind Kernal-ROM, beide BASIC-ROMs und der Zeichengenerator sichtbar. Als RAM-Bank ist Bank 0 ausgewählt.
PCRD ($d504) =	$41, entspricht PCRC mit RAM-Bank 1.

1.1.3 Das Mode-Konfigurationsregister (MCR, $d505)

Dieses Register der MMU kontrolliert in der Hauptsache, welcher der beiden Prozessoren des C128 Zugriff auf den Speicher erhält. Ist das Bit 0 des Registers gleich 0, ist die Z80 aktiv, ist es 1, ist unser gewohnter 8502 aktiviert.
Im Fall, daß die 8502 aktiv ist, entscheidet das Bit 6 des MCR, in welchem Modus gearbeitet wird. Bei gesetztem Bit 6 befindet man sich im 64er-Modus, bei gelöschtem Bit 6 im 128er-Modus.
In Bit 7 schließlich wird die Information aufbewahrt, ob der Computer an einen 40- oder einen 80-Zeichen-Schirm angeschlossen ist. Dazu wird bei einem RESET oder beim Einschalten der Zustand der 40/80-DISPLAY-Taste abgefragt. Gelöschtes Bit 7 zeigt an, daß die Taste gedrückt war.
Die Bits 1, 2, 4 und 5 haben im 128er-Modus keine Bedeutung. Bit 3 wird als Kontrollbit dafür benutzt, ob eine schnelle serielle Datenübertragung zur Floppy erfolgen kann, dies im Fall, daß eine 1570 oder 1571 angeschlossen ist.

1.1.4 Das RAM-Konfigurationsregister (RCR, $d506)

Dieses Register ist wieder sehr interessant für den Programmierer. In ihm kann festgelegt werden, in welcher Speicherbank die Bildschirmspeicher des VIC II liegen sollen. Dafür ist die obere Hälfte des Registers vorgesehen, wobei in der vorliegenden Version des C128 nur die Bits 7 und 6 relevant sind:

Bit 7	Bit 6	RAM-Bank für Video-RAM und Grafik
0	0	0
0	1	1
1	0	2
1	1	3

In der Grundversion des C128 mit zwei Bänken ist Bit 7 wieder ohne Bedeutung. Die Normaleinstellung der VIC-Bank ist 0, das Video-RAM liegt also bei $400 in Bank 0.
Ein leicht vorstellbarer Einsatz dieses Registers wäre zum Beispiel das schnelle Hin- und Herschalten zwischen zwei Bildschirmen, die an der gleichen Adresse, aber in verschiedenen Speicherbänken liegen. Für Spiele und ähnliches ist dies in Verbindung mit der Möglichkeit eines Rasterzeilen-IRQ sicher eine interessante Möglichkeit.
Die wichtigere Hälfte des MCRs sind die Bits 3 bis 0. Durch sie können Bereiche des 64K-Adreßraums der CPU zum sogenannten gemeinsamen Bereich (Common Area) deklariert werden. Unter einem gemeinsamen Bereich versteht man einen RAM-Bereich, der von der möglichen Umschaltung durch das CR der MMU einfach nicht betroffen wird. Einerlei, welche Konfiguration im Konfigurationsregister eingestellt wird, der gemeinsame Bereich bleibt unverändert. Der Einfachheit halber wird im gemeinsamen Bereich immer das RAM der Bank 0 angesprochen.
Der Sinn und Zweck einer solchen Einrichtung wird sofort klar, wenn man sich einmal ansieht, was wäre, wenn es ihn nicht gäbe:
Stellen wir uns dazu einmal vor, wir benötigten in einem Programm einen Wert, der in einer anderen Speicherbank abgelegt ist. Wir laden also das Konfigurationsregister mit dem Wert, der die gewünschte Speicherbank auswählt und stellen im gleichen Augenblick fest, daß unser Programm abgestürzt ist. Wie das? Nun, durch die Umschaltung auf eine andere Bank haben wir unser Programm ja selbst für den Prozessor unsichtbar gemacht, der Absturz war also praktisch vorprogrammiert. Gleichzeitig können wir aus diesem Beispiel erkennen, daß Programm und Daten offensichtlich immer gleichzeitig vom Prozessor einsehbar sein müssen. Dies wird durch einen gemeinsamen Bereich möglich, in dem entweder die Daten stehen oder das Programm, das Daten aus anderen Bänken holt.
Das MCR erlaubt die Einrichtung von maximal zwei Common Areas, die am Speicheranfang und am Speicherende stehen können. Die Größe der gemeinsamen Bereiche kann zwischen 1 und 16 KByte gewählt werden:

Bit 3	Bit 2	gemeinsamer Bereich
0	0	keiner
0	1	Common Area von $0000 nach oben
1	0	Common Area von $ffff abwärts
1	1	sowohl oben als auch unten eine Common Area

Bit 1	Bit 0	Größe der Common Area(s)
0	0	1 KByte
0	1	4 KByte
1	0	8 KByte
1	1	16 KByte

BASIC 7.0 benutzt eine Common Area, die im unteren Bereich bis $400, also bis zum Beginn des VIC II-Video-RAMs reicht. In diesem Raum liegen Daten – man denke an die Zeropage, Stack usw. – aber auch die später noch besprochenen Unterprogramme von BASIC und Betriebssystem, die Werte aus anderen Konfigurationen holen, beziehungsweise in andere Konfigurationen schreiben.

1.1.5 Die Pagepointer (P0L, P0H, P1L, P1H, $d507-$d50a)

Diese Register eröffnen dem Programmierer bisher unbekannte, ziemlich exotisch anmutende Möglichkeiten. Mit ihrer Hilfe kann eine Verlegung von Zeropage und Prozessorstack vorgenommen werden! Welche Auswirkungen dies auf verschiedene Problemlösungen haben wird, bleibt noch abzuwarten.
Die Register P0L und P1L bestimmen, in welchem 256-Byte-Block die neue Zeropage beziehungsweise der neue Stack liegen soll, von den Registern P0H und P1H sind für uns nur die Bits 0 und 1 interessant, die angeben, aus welcher Speicherbank der 256er-Block entnommen wird.

Bit 1	Bit 0	RAM-Bank
0	0	0
0	1	1
1	0	2
1	1	3

Soll sowohl die RAM-Bank als auch die Adresse von Zeropage oder Stack verändert werden, muß immer zuerst das Register beschrieben werden, das die RAM-Bank festlegt. Dieses wird dann solange zwischengespeichert, bis auch das korrespondierende Register mit der neuen Adresse beschrieben wurde.
Technisch wird die Verlegung so durchgeführt, daß die MMU nach der Neueinstellung der Zeropage bei jedem Zeropage-Zugriff den 256-Byte-Block anwählt, der durch P0L und P0H eingestellt ist. Andersherum wird jeder Zugriff des Prozessors auf eine Adresse innerhalb der neuen Zeropage auf die alte Zeropage umgebogen. Alte Zeropage und der neu eingestellte 256-Byte-Block tauschen damit praktisch ihre Plätze, ohne daß der Prozessor etwas davon bemerkt. Die Verlegung des Stacks erfolgt auf die gleiche Art. Der

Block, mit dem Zeropage und Stack den Platz tauschen, kann auch in anderen RAM-Bänken liegen. Hier ist allerdings Vorsicht geboten, denn wie man mit den folgenden kleinen Programmen leicht nachprüfen kann, wird die Verlegung in eine andere RAM-Bank erst dann korrekt ausgeführt, wenn Zeropage und Stack nicht in einer Common Area liegen:

```
lda #$00    I/O sichtbar
sta $ff00   machen
lda #$01    Pagepointer 0
sta $d508   auf Bank 1,
lda #$0c    Adresse $0c00
sta $d507
lda #$45    ASCII-'e'
sta $02     in neue Zeropage
lda #$00    alte Werte des
sta $d508   Pagepointers
sta $d507   wiederherstellen
brk
```

Dieses Programm, mit Hilfe des Monitors eingegeben, sollte die Zeropage an die Adresse $c00 in Bank 1 verlegen. Nach Ablauf des Programms müßte deshalb an der Stelle $c02 ein ASCII-'e' stehen. Tut es aber nicht! Vielmehr steht das gesuchte Zeichen in Bank 0 bei $c02. Schaltet man dagegen die Common Area bei der Verlegung der Zeropage ab, klappt es:

```
lda #$00    I/O sichtbar
sta $ff00   machen
lda $d506   RAM-Configuration-Register
and #$f3    Bits 3 und 2 löschen =
sta $d506   Common Area ausschalten
lda #$01    Zeropage wie vorher zu
sta $d508   verlegen versuchen
lda #$0c
sta $d507
lda #$45    und ASCII-'e'
sta $02     in neue Zeropage
lda #$00    alte Werte
sta $d508   wiederherstellen
sta $d507
lda $d506
ora #$04
sta $d506
rts
```

1.1.6 Das Versionsregister der MMU (VR, $d50b)

Dieses Register erfüllt eigentlich keine Funktion im Sinne der Speicherverwaltung. Im VR kann lediglich durch Anwenderprogramme festgestellt werden, wieviele RAM-Bänke in den C128 eingebaut sind, und welche Versionsnummer die eingebaute MMU hat. Das Register kann nicht beschrieben werden. Wird es ausgelesen, enthält das höherwertige Halbbyte die Anzahl verfügbarer RAM-Bänke, das niederwertige Halbbyte enthält die Versionsnummer der MMU.

1.2 Der 80-Zeichen-Videocontroller (VDC, VDCST $d600, VDCDAT $d601)

Zu den großen Vorzügen, die der C128 gegenüber seinem Vorgänger, dem C64, zu bieten hat, gehört neben dem vergrößerten Arbeitsspeicher auch die Möglichkeit, endlich professionell mit einem 80-Zeichen-Bildschirm arbeiten zu können. Textverarbeitung zum Beispiel war ja auf dem C64 immer eine Sache, an der sich die Geister schieden, weil es einfach nicht möglich war, eine Textzeile im Ganzen auch auf dem Bildschirm darzustellen.
Die Darstellung des 80-Zeichen-Schirms übernimmt im 128er der sogenannte VDC, der im Speicher des C128 nur an zwei Adressen, $d600 und $d601 im I/O-Bereich, vertreten ist. Jeder Datentransfer von und zum VDC muß über diese beiden Adressen abgewickelt werden.
Der VDC verfügt über 16 KByte eigenes RAM, das vom Prozessor nicht eingesehen werden kann, genausowenig, wie der VDC aus den Speicherbänken der CPU lesen kann. Dies unterscheidet den VDC grundsätzlich vom VIC II, der praktisch als Coprozessor der jeweiligen CPU fungiert. In den 16 KByte des VDC-Speichers werden sowohl das 80-Zeichen-Video-RAM als auch das dazugehörende Attribut-RAM, eine leistungsfähigere Entsprechung zum Farb-RAM des VIC II, untergebracht. Da der VDC keinerlei Verbindung mit dem Hauptspeicher besitzt, ist es unumgänglich, auch den gesamten Zeichenvorrat des 128ers in den VDC-Speicher zu übertragen. Daraus ergibt sich als erfreulicher Nebeneffekt, daß auf dem 80-Zeichen-Schirm beide Zeichensätze, die im 40-Zeichen-Modus durch Drücken von C= und Shift erreicht werden, gleichzeitig dargestellt werden können.
Das VDC-RAM kann nicht direkt beschrieben werden, es kann auch nicht direkt durch VDCST und VDCDAT ($d600 und $d601) adressiert werden. Jeder Zugriff auf das VDC-RAM erfolgt über die Register des Videocontrollers.

Um ein Register des VDC beschreiben und lesen zu können, wird als erstes immer die Nummer des Registers in VDCST abgelegt:

```
lda #nr
sta $d600
```

Auf das Beschreiben von VDCST mit einer Registernummer reagiert der VDC damit, daß der Inhalt des gewünschten Registers nach VDCDAT übertragen wird, von wo er ausgelesen werden kann.
Um ein Register beschreiben zu können, muß man wissen, daß das Bit 7 des VDCST ein READY-Flag des VDC ist, mit dem der Videocontroller anzeigt, daß er zur Aufnahme von Daten bereit ist. Bevor ein Beschreiben von VDCDAT mit einem neuen Registerwert erfolgen kann, muß also zunächst das Bit 7 der Adresse $d600 abgefragt und so lange gewartet werden, bis der VDC seine Bereitschaft anzeigt:

```
marke   lda $d600
        bpl marke
```

Wer ganz auf Nummer Sicher gehen will, kann schon vor der Eingabe der Registernummer nach VDCST prüfen, ob der VDC bereit ist. In 99% der Fälle ist dies jedoch unnötig, da der VDC schnell genug arbeitet.
Sehen wir uns zunächst den internen Aufbau des VDC-Speichers an, bevor die 37 Register des VDC und ihre Benutzung näher beschrieben werden. Die Normalbelegung des VDC-RAM ist:

0- $7ff	Video-RAM
$800- $fff	Attribut-RAM
$2000-$3fff	Zeichensätze

Bei dieser Belegung fällt auf, daß der Bereich von $1000 bis $1fff noch zu vergeben ist. Wir haben also noch 4 KByte Spielraum innerhalb des VDC-RAM, den man nutzen kann (s.u.). Aus technischen Gründen belegt jede Zeichendefinition im VDC-RAM 16 statt wie im Character-ROM 8 Byte, daher werden für die normalerweise 4 KByte Zeichensätze beim VDC 8 KByte RAM benötigt. Die oberen 8 der 16 Byte Zeichendefinition sind einfache Nullbytes.
Der Zusammenhang zwischen dem Bildcode eines Zeichens, der im Video-RAM abgelegt wird, und der Adresse der zugehörigen Zeichendefinition im VDC-RAM ergibt sich somit zu:

Adresse = Bildcode*16 + $2000

1.2.1 Das Attribut-RAM ($800-$fff)

Wie schon erwähnt, stellt das Attribut-RAM des VDC eine Entsprechung zum Farb-RAM des VIC II dar. Es kommen allerdings einige Optionen zur reinen Farbgebung hinzu, die das Attribut-RAM wesentlich interessanter machen als das Farb-RAM, deshalb das eigene Unterkapitel.
Im unteren Halbbyte, Bits 3 bis 0, wird die Schriftfarbe für jedes einzelne Zeichen festgelegt. Die Aufschlüsselung der einzelnen Farbcodes ist:

Bit 3	Bit 2	Bit 1	Bit 0	Farbe
0	0	0	0	schwarz
0	0	0	1	dunkelgrau
0	0	1	0	blau
0	0	1	1	hellblau
0	1	0	0	dunkelgrün
0	1	0	1	hellgrün
0	1	1	0	stahlblau
0	1	1	1	türkis
1	0	0	0	rot
1	0	0	1	hellrot
1	0	1	0	violett
1	0	1	1	lila
1	1	0	0	braun
1	1	0	1	gelb
1	1	1	0	hellgrau
1	1	1	1	weiß

Das obere Halbbyte, Bits 7 bis 4, enthält weitere Informationen zur Zeichendarstellung:

Bit 7 (ALT)

Dieses Bit bestimmt, aus welchem der beiden Zeichensätze die Zeichendarstellung entnommen werden soll. Ist das Bit gesetzt, wird die Darstellung gemäß des zweiten Zeichensatzes (Groß-/Kleinschreibung) vorgenommen.

Bit 6 (REV)

Dieses Bit bewirkt die reverse Darstellung eines Zeichens, wenn es gesetzt ist. Von dieser Option macht der Kernal-Bildschirm-Editor übrigens keinen Gebrauch. Wohl um das Programm kurz zu halten und Rechenzeit einzusparen, werden statt dessen die reversen Zeichensätze mit in das VDC-RAM kopiert.

Bit 5 (UNDL)

Gesetztes Bit 5 führt dazu, daß das Zeichen unterstrichen dargestellt wird. Die Rasterzeile, also die Zeile in der Punktmatrix der Zeichendefinition, in der der Unterstreichungsstrich stehen soll, kann außerdem im Register 29 noch frei gewählt werden.

Bit 4 (FLASH)

Ist das FLASH-Bit gesetzt, blinkt das zugehörige Zeichen. Die Blinkfrequenz läßt sich in Register 24 in zwei Stufen wählen.
Die einzelnen Attribut-Bits können selbstverständlich auch in Kombinationen miteinander verwendet werden.
Welche Möglichkeiten sich allein aus der Verwendung der Attribute in eigenen Programmen ergeben, kann man sich leicht ausmalen. Man denke nur an Textverarbeitungen, die einen unterstrichenen Satz auch wirklich unterstrichen darstellen können, oder an die Nutzung der reversen Zeichensätze für die Definition eigener Zeichen zusätzlich zum normalen Zeichensatz, indem man das REV-Bit ausnützt.

1.2.2 Register des VDC

Nicht jedes Register des VDC läßt sich in eigenen Programmen sinnvoll nutzen, der VDC mit seiner Vielzahl von Einstellmöglichkeiten lädt aber geradezu dazu ein, mit ihm zu experimentieren. Sie werden sicher überrascht sein, was sich mit dem VDC alles anstellen läßt.

Register 0

Dieses Register enthält die Anzahl Zeichen pro Bildschirmzeile, die inklusive des notwendigen Strahlrücklaufs zur nächsten Zeile dargestellt werden müßten. Die Normaleinstellung für diese Konstante ist $7e (126). Für den Programmierer ist dieses Register von keiner weiteren Bedeutung.

Register 1 (Zeichen/Zeile)

Zeilenbreite in Zeichen. Diese ist normalerweise gleich 80, es könnte sich aber die Notwendigkeit ergeben, den Wert zu ändern, wenn man die Zeichenbreite verändert.

Register 2

Hier kann der linke Rand des Bildschirms auf dem Monitor eingestellt werden, wenn man einen anderen als den 1901-Monitor betreibt. Der normale Wert des Registers ist $66 (102).

Register 3
Auch ein technischer Wert, der angibt, wie lange der Strahlrücklauf horizontal und vertikal dauert. Standardwert ist $49 (73).

Register 4
Register 4 entspricht dem Register 0 für die Anzahl der dargestellten Zeilen. Normaler Wert des Registers ist $27 (39).

Register 5
Feineinstellung zu Register 4. Normalwert $e0 (224), wobei die oberen 3 Bit immer gesetzt sind.

Register 6 (Anzahl Zeilen)
Dies ist wieder ein interessantes Register. In Entsprechung zu Register 1 kann hier die Anzahl der Bildschirmzeilen verändert werden (normal 25). So hinzugewonnene Zeilen können natürlich nicht durch die Kernal-Routinen genutzt werden, denn diese sind nur auf 25 Zeilen ausgelegt. Sollte man die Zeilenanzahl vergrößern wollen, ist darauf zu achten, daß gegebenenfalls die internen Adressen von Video-RAM und Attribut-RAM entsprechend angepaßt werden (Register 12/13 und 20/21).

Register 7
Einstellung des oberen Bildschirmrandes entsprechend zu Register Nummer 2. Standard ist $20 (32).

Register 8
Normalwert ist $fc (252). In den Bits 0 und 1 dieses Registers kann eine sogenannte Anzeigeform eingestellt werden, deren Sinn ich, ehrlich gesagt, nicht erkenne. Am besten, Sie probieren die einzelnen Einstellungen einmal aus.

Register 9 (Zeichenlänge)
Anzahl der Rasterzeilen pro Zeichen –1, sprich Zeichenlänge. Normalwert ist für dieses Register $e7 (231). In Verbindung mit der Definition der Zeichenzeilenlänge (Register 23) ergibt sich die Möglichkeit, den vertikalen Abstand zweier Zeichenzeilen voneinander zu verändern. Die Bits 7 bis 5 dieses Registers sind immer gesetzt.

Register 10 (Cursor-Darstellung)
Darstellung des Cursors. Die Bits 5 und 6 bestimmen, ob der Videocontroller die Anzeige des Cursors übernehmen soll. Dabei bedeuten:

Bit 6	Bit 5	
0	0	Cursor ist starr
0	1	Cursor ist aus
1	0	Cursor blinkt schnell
1	1	Cursor blinkt langsam

Die Normaleinstellung von $a0 (160) bedeutet, daß der Cursor langsam blinkt.
Die Bits 4 bis 0 legen den Start des Cursors innerhalb der Punktmatrix eines Zeichens fest. Bit 7 ist immer gesetzt.

Register 11 (Ende des Cursors)
In diesem Register wird in den Bits 4 bis 0 entsprechend zu Register 10 das Ende des Cursors festgelegt. Normalerweise enthält Register 11 den Wert $e7 (231), der normale Block-Cursor reicht also von der Punktmatrixzeile 0 bis 7 und überdeckt damit das gesamte Zeichen.

Register 12+13 (Start des Video-RAMs)
Startadresse des Video-RAMs innerhalb des VDC-RAMs. Register 12 enthält das Highbyte, Register 13 das Lowbyte. Normalerweise beginnt das Video-RAM bei $0000, beide Register enthalten also eine Null. Um den Umgang mit dem VDC noch etwas zu illustrieren, möchte ich bei Gelegenheiten wie dieser immer kleine Beispielprogramme bringen, die zwar nicht sofort nutzbringende Anwendungen darstellen, aber meiner Erfahrung nach eine raschere Orientierung über Sachverhalte erlauben, wenn man etwas nachschlagen will. Wie geht man also vor, wenn man das Video-RAM verschieben will?

```
        lda #$00     I/O-Bereich sichtbar
        sta $ff00    machen
m1      bit $d600    sicherheitshalber prüfen, ob
        bpl m1       der VDC zum Empfang bereit ist
        ldx #$0c     Register 12
        stx $d600    anwählen
m2      bit $d600    und warten, bis der VDC zum
        bpl m2       Datenempfang bereit ist
        lda #...     neue Startadresse Highbyte
        sta $d601    übertragen
m3      bit $d600    warten, bis die Aktion
        bpl m3       abgeschlossen ist
        inx          Register 13
        stx $d600    anwählen
m4      bit $d600    und warten
        bpl m4
        lda #...     Lowbyte der neuen Startadresse
        sta $d601    übertragen
```

Register 14+15 (Cursor-Position)
Aktuelle Cursor-Position. Das Register 14 enthält wieder das Highbyte und das Register 15 das Lowbyte der Cursor-Adresse innerhalb des Video-RAMs. Diese Adresse muß angegeben werden, wenn der Videocontroller den Cursor selbständig blinken lassen soll.

Register 16+17 (Lichtgriffel)

Dieses Registerpaar enthält bei Benutzung eines Lightpen die vertikale beziehungsweise horizontale Adresse des Lightpen. Die Bits 7 und 6 des Registers 16 sind immer 0, in den anderen Bits wird die Lightpen-Position in Zeichenzeilen+1 angegeben. Register 17 enthält die Position in Zeichenzeilen+8.

Register 18+19 / 31 (Speicheradresse)

High- und Lowbyte der Adresse innerhalb des Video-RAM, auf die der nächste Zugriff erfolgen soll – mit diesen Registern kann also das VDC-RAM adressiert werden. Register 31 ist das Datenregister hierzu, in das man die zu schreibenden Daten bringt. Wird das VDC-RAM ausgelesen, stehen in Register 31 die gelesenen Bytes.
Die Übernahme eines zu schreibenden Bytes aus Register 31 in das VDC-RAM erfolgt nach dem Setzen der Adresse durch einen Zugriff auf eines der Register 18 oder 19. Um beispielsweise den Bildcode von a an die Adresse $1c3 des Video-RAM zu schreiben, muß man folgendermaßen vorgehen:

```
     lda #0       I/O sichtbar
     sta $ff00    machen
     ldx #18      Register 18
     lda #1       mit dem Highbyte der Adresse laden
     jsr set      (UP zur Übergabe der Werte an den VDC)
     inx          Register 19 mit dem
     lda #$c3     Lowbyte laden
     jsr set
     lda #1       Bildcode von a
     ldx #31      nach Register 31 bringen
     jsr set
     ldx #18      Zur Übernahme des Datums aus Register
     jsr set      31 in das RAM-Register 18 nochmals
     rts          ansprechen
set  stx $d600    UP zur Übergabe von Registerwerten
m1   bit $d600    READY des VDC abwarten
     bpl m1
     sta $d601    dann VDCDAT beschreiben
     rts
```

Bei der internen Blockverschiebung dient das Registerpaar 18/19 zur Festlegung der Zieladresse.

Register 20+21 (Start des Attribut-RAMs)

Das Registerpaar 20 und 21 entspricht den Registern 12 und 13 für das Attribut-RAM. Der normale Startwert für das Attribut-RAM ist $800.

Register 22+23

Register 22 enthält im oberen Halbbyte, Bits 7 bis 4, die Zeichenbreite inklusive des horizontalen Zeichenzwischenraums, ausgedrückt in Pixel −1. Das untere Halbbyte enthält die Zeichenbreite ohne den Zwischenraum. Mit diesem Register kann also gleichzeitig die Zeichenbreite und der horizontale Zeichenzwischenraum definiert werden.
Register 23 enthält im unteren Halbbyte mit Bit 4 entsprechend die Zeichenlänge mit vertikalem Zwischenraum. Register 9 ist die Entsprechung zum unteren Halbbyte des Registers 22.
Die Standardwerte für die betroffenen Register sind:

Reg. 9	$e7	Zeichenlänge = 8 Pixel
Reg. 23	$e8	dargestellte Rasterzeilen = 9 (1 Rasterzeile vertikaler Zwischenraum)
Reg. 22	$78	Zeichenbreite = 8, dargestellte Rasterspalten = 9, Abstand horizontal = 1

Register 24 (Zeichenbreite)

Die Bits 0 bis 4 dieses Registers sind für das weiche vertikale Scrollen nach oben zuständig. Werden die Bits mit einer Zahl zwischen 0 und der Zeichenlänge (Register 9) beschrieben, so wird der Bildschirm um den Inhalt an Rasterzeilen nach oben verschoben dargestellt. Beschreibt man diese Bits nacheinander mit den Werten von 1 bis zur Zeichenlänge oder umgekehrt, ergibt sich der Scroll-Effekt nach oben beziehungsweise unten.
Bit 5 des Registers steuert die Blinkfrequenz für diejenigen Zeichen, in deren Attribut das FLASH-Bit gesetzt ist. Ist Bit 5 gleich 0, beträgt die Blinkfrequenz $1/16$ der Bildwiederholfrequenz, ansonsten wird die Blinkfrequenz halbiert.
Bit 6 schaltet den gesamten Bildschirm auf reverse Darstellung, wenn es gesetzt ist.
Bit 7 erhält seine Bedeutung in Zusammenhang mit Blockoperationen (siehe Register 32 und 33). Ist Bit 7 gesetzt, wird ein Kopiervorgang eingeleitet, ist es 0, wird ein Speicherbereich des VDC-RAMs aufgefüllt.

Register 25 (Steuerregister 1)

Das untere Halbbyte dient zum horizontalen Scrollen nach links, entsprechend zum vorigen Register.
Das Bit 4 verdoppelt die dargestellte Zeichenbreite, wenn es gesetzt ist.
Bit 5 entscheidet über die Farbe des horizontalen Zeichenzwischenraums. Ist es gesetzt, wird der Zwischenraum mit der Zeichenfarbe gefüllt, ist es 0, hat der Zwischenraum die Farbe des Hintergrundes.

Bit 6 bestimmt, ob der Bildschirm mit einheitlicher Schriftfarbe betrieben wird, die dann in Register 26 angegeben werden muß (Bit 6 = 0), oder ob die Schreibfarbe dem Attribut-RAM entnommen wird (Bit 6 = 1).
Bit 7 schließlich schaltet den Grafikmodus des VDC ein, wenn es gesetzt wird. Im Einzelpunktmodus werden die gesamten 16 KByte des VDC-RAMs zur Darstellung der Bitmap gebraucht. Jedes gesetzte Bit der Bitmap entspricht einem sichtbaren Pixel auf dem Bildschirm. Die Darstellung ist monochrom, die Farbinformation stammt aus Register 26. Der logische Aufbau des Grafikschirms ist einfacher als beim VIC II einzusehen. Jedes Byte der Bitmap beschreibt 8 horizontal nebeneinanderliegende Pixel, das Byte $0000 enthält in Bit 7 die obere linke Ecke des Grafikschirms. Der 9. Punkt der ersten Grafikzeile befindet sich dann in Bit 7, Byte $0001, der 17. in Byte $0002 usw. immer Zeile für Zeile fortlaufend. Jede Grafikzeile ist 80 Byte lang, insgesamt besteht der Grafikschirm aus 200 solcher Zeilen. Der Zusammenhang zwischen Zeile und Spalte eines Grafikpunktes und seiner Adresse im VDC-RAM ist durch diesen einfachen Aufbau ebenfalls sehr einfach. Er läßt sich darstellen durch:

Adresse = int(Spalte/8) + 80*Zeile

Zeile und Spalte werden dabei mit Null beginnend gezählt. Der angesprochene Punkt innerhalb des Bytes ergibt sich ebenso einfach zu:

Pixel = 2 ↑ (7 – (Spalte und 7))

Register 26 (Farben)

Dies ist das Farbregister für den Grafik- und den monochromen Modus (Register 25, Bits 7 beziehungsweise 6). Das obere Halbbyte enthält die Schrift- oder Pixelfarbe, das untere die Hintergrundfarbe. Die Codierung der Farben entspricht der im Attributspeicher.

Register 27 (Adressen-Inkrement)

Der Inhalt des Registers 27 wird beim Übergang von einer Bildschirmzeile in die nächste automatisch zur Video-RAM- und Attribut-RAM-Adresse hinzuaddiert. Das kann nützlich sein, wenn zum Beispiel durch das Setzen des Bits 4 in Register 25 die Zeichen doppelt breit ausgegeben werden. Wird dann das Register 27 auf 40 gesetzt, synchronisiert dies die Bildschirmzeilenanfänge und die Anfänge der Zeilen im Video-RAM. Normalwert dieses Registers ist Null.

Register 28 (Start des Zeichensatzes)

Auch die Lage der Zeichendefinitionen innerhalb des VDC-Speichers kann verändert werden. Die Bits 7 bis 5 dieses Registers stellen die drei höchstwertigen Bits der Startadresse des Character-RAMs dar. Normalerweise ist nur Bit 5 gesetzt, das Zeichen-RAM liegt folglich bei $2000.

Bit 4 enthält die Information, welche Art von RAM-Bausteinen das VDC-RAM bilden. Dies ist eine rein technische Angabe, die nicht verändert werden sollte.
Normalwert für dieses Register: $2f (47).

Register 29 (Position Unterstreichung)
Hier kann man in den Bits 4 bis 0 bestimmen, in welcher Zeile der Punktmatrix der Unterstreichungsstrich stehen soll. Gemessen wird die Zeile in Pixel – 1. Der Standardwert dieses Registers von $e7 legt den Unterstreichungsstrich in die Rasterzeile 8. Neueinstellungen des Registers können vor allem dann notwendig werden, wenn die Zeichenlänge in den anderen Registern verändert wurde.

Register 30 (COUNT)
Dies ist ein Zählregister für die sogenannten Blockoperationen, die der VDC selbständig vornehmen kann. Blockoperationen sind Blockverschiebungen innerhalb des VDC-RAMs und das Füllen von Speicherbereichen mit einem Wert.
Register 30 enthält bei solchen Operationen die Anzahl der zu verschiebenden beziehungsweise zu schreibenden Bytes. Gleichzeitig bewirkt das Beschreiben des Registers den Start der jeweiligen Operation (s. bei Registern 32/33).

Register 31 (DATA)
Das Datenregister des VDC wurde schon mehrfach angesprochen. Jeder Zugriff von außerhalb auf das RAM des VDC erfolgt über das Datenregister in Verbindung mit den Registern 18 und 19. Beim gerade angesprochenen Füllen von Speicherbereichen steht das Füllbyte im Datenregister.

Register 32+33 (Blockstart)
Hier steht im Fall der VDC-internen Blockverschiebung die Ursprungsadresse des zu verschiebenden Blocks. Wie immer bei Registerpaaren des VDC enthält das Register mit der niedrigeren Nummer das höherwertige Byte.
Da jetzt alle für die Blockoperationen notwendigen Register beisammen sind, sollen jetzt zwei Beispiele folgen, die zeigen, wie man damit umgeht:
Zunächst wird immer durch Bit 7 des Registers 24 die Betriebsart festgelegt.

Bit 7, Register 24
0 Füllen
1 Blockverschiebung

Füllen eines Bereiches: Es soll die erste Zeile des Video-RAMs mit Leerzeichen gefüllt werden.

```
        lda #0         I/O sichtbar
        sta $ff00      machen
        ldx #24        Register 24
        stx $d600      anwählen
m1      bit $d600      warten, bis ready
        lda $d601      Registerwert auslesen
        and #$7f       Bit 7 löschen
        jsr set        und wieder nach Register 24
        ldx #18        Register 18
        lda #0         mit Lowbyte der zu füllenden Adresse
        jsr set        laden
        inx            Register 19 mit dem Highbyte
        jsr set
        lda #$20       Leerzeichen
        ldx #31        ins Dataregister
        jsr set
        lda #80        für 80 Zeichen
        dex            ins Zählregister und gleichzeitig
        jsr set        Start der Operation
        rts

set     stx $d600      Register anwählen
m2      bit $d600      warten, bis ready
        bpl m2
        sta $d601      dann Wert übergeben
        rts
```

Verschieben eines Bereiches: Wir schieben Zeile 1 zur Zeile 0. Die Startadresse der zweiten Zeile ist gleich 80.

```
        lda #0         I/O sichtbar
        sta $ff00      machen
        ldx #18        als Zieladresse die 0 in das
        jsr set        Registerpaar 18/19
        inx
        jsr set
        ldx #32        Highbyte der Quelladresse ist
        jsr set        auch Null
        lda #80        Lowbyte der Quelladresse
        inx            nach Register 33
        jsr set
```

```
        ldx #24       Register 24
        stx $d600     auswählen
m3      bit $d600     und warten, bis ready
        bpl m3
        lda $d601     dann den Wert auslesen und
        ora #$80      Bit 7 setzen
        jsr set       Register 24 beschreiben
        lda #80       80 zu übertragende Byte
        ldx #30       in den Zähler
        jsr set       speichern und Start der Verschiebung
        rts
```

Register 34 bis 36

Die letzten drei VDC-Register haben für den Programmierer wieder wenig Wert. Sie enthalten technische Angaben, zum Beispiel wie oft die dynamischen RAMs aufgefrischt werden müssen etc. Die Standardwerte dieser Register sind $7d (125), $40 (64) und $f5 (245).

1.3 Der 40-Zeichen-Videocontroller (VIC II, $d000)

Wahrscheinlich aus Gründen, die in der verlangten Kompatibilität des 128er-Systems mit dem C64 liegen, enthält der 128er außer einem 80-Zeichen-Chip auch den alten Videocontroller des C64, der nur mit der langsameren Taktfrequenz von 1 MHz (SLOW) betrieben werden kann. Inwieweit die durch das Vorhandensein zweier Controller auftretenden Nebeneffekte, wie die Möglichkeit, Text und Grafik auf getrennten Bildschirmen darstellen zu können, eine gewollte Bereicherung der Fähigkeiten des 128ers sind, oder ob sie vielmehr reine Nebeneffekte darstellen, kann sich jeder selbst beantworten. Man könnte sogar aus der Tatsache, daß die Grafikbefehle des BASIC sich auf den VIC II beziehen, schließen, daß der 80-Zeichen-Chip nachträglich eingeplant wurde. Nun, wie dem auch sei, mit Sicherheit wird der C128 durch seine zwei Videocontroller nicht uninteressanter, eher das Gegenteil.
Da der VIC II des 128ers so weitgehend mit dem des C64 übereinstimmt, und weil die Möglichkeiten, die dieser Chip bietet, schon weitgehend von BASIC-Befehlen ausgenutzt werden, soll an dieser Stelle eine einfache Beschreibung erfolgen, die zum Verständnis der Befehle auf Maschinenspracheebene ausreicht.
Einzige Ergänzung des VIC II gegenüber seinen gleichnamigen Vorläufern sind die neuen Register 47 und 48, in dessen Bit 0 die Taktfrequenz gewählt werden kann.

1.3.1 Register des VIC II

Register 0 (Sprite-Koordinate horizontal)

Sprite Nummer 0, Koordinate in Rasterspalten, Bits 0 bis 7. Die Koordinaten der Sprites sind nicht mit den gewöhnlichen Koordinaten des Grafikschirms identisch, da Sprites auch über das Bildfenster hinaus beweglich sind. Der Nullpunkt des Sprite-Koordinatensystems liegt zwar auch links oben, der Nullpunkt des Grafikschirms liegt darin aber schon bei 20 Pixel horizontal und 30 vertikal. Die rechte untere Ecke des Bildschirms hat die Koordinaten (340,220). Maximalkoordinaten sind (511,255). Da die Koordinaten in horizontaler Richtung größer als 255 werden können, ist ein weiteres Bit notwendig, das für alle Sprites in Register 16 aufbewahrt wird.

Register 1 (Sprite-Koordinate vertikal)

Sprite Nummer 0, vertikale Koordinate in Rasterzeilen, entsprechend zu Register 0.

Register 2+3 / 4+5 / ...– 14+15

Diese Registerpaare entsprechen den Registern 0 und 1 jeweils für die Sprites mit Nummern 1 bis 7.

Register 16

Hier werden die höchstwertigsten Bits der horizontalen Koordinaten der Sprites aufbewahrt. Bit 0 ist das neunte Bit der Koordinate des Sprites 0, Bit 1 gehört zu Sprite 1 usw.

Register 17 (VIC-Steuerregister 1)

Die Bits 2 bis 0 verschieben den oberen Bildschirmrand, mit ihnen kann ein weiches Scrollen in vertikaler Richtung realisiert werden.
Mit Bit 3 wird der VIC von 24 Zeilen (=0) auf 25 Zeilen (=1) umgeschaltet.
Bit 4 schaltet den VIC ein und aus. Ist es gleich Null, wird der Bildschirm abgeschaltet.
Bit 5 schaltet den Grafikmodus ein, wenn es gesetzt wird. Bit 6 schaltet den sogenannten Extended-Color-Mode ein, einen farbigen Textmodus.
Bit 7 gehört als neuntes Bit zu Register 18.

Register 18 (Koordinate für Rasterzeilen-IRQ)

Vertikale Koordinate für die Auslösung eines Rasterzeilen-IRQs in Pixel. Dazu wird noch ein Koordinatensystem herangezogen, in dem der obere Rand des Monitors bei 30 Pixel liegt, der untere bei 280 Pixel. Diese Werte können von Monitor zu Monitor leicht variieren. Ist die Option eines Rasterzeilen-IRQ gewählt, wird bei jedem Aufbau der hier angegebenen Rasterzeile durch den VIC die Arbeit des Prozessors unterbrochen und gegebenenfalls eine Aktion ausgelöst. Vom BASIC wird der Rasterzeilen-IRQ dazu genutzt, den bekannten, in einen Grafik- und einen Textteil aufgesplitteten Bildschirm

herzustellen. Dazu wird der VIC bei jedem Rasterzeilen-IRQ entweder vom Text- in den Grafikmodus oder umgekehrt umgeschaltet.

Register 20+21 (Lichtgriffel)
Horizontale beziehungsweise vertikale Position des Kathodenstrahls, als der Lightpen ein Signal abgab. Hier muß natürlich noch die Zeitverzögerung mit eingerechnet werden, denn der Strahl ist sehr schnell.

Register 21 (Sprite ein/aus)
Jedem Bit dieses Registers ist ein Sprite mit der gleichen Nummer zugeordnet. Das betreffende Sprite wird durch Setzen des korrespondierenden Bits eingeschaltet.

Register 22 (VIC-Steuerregister 2)
Steuerregister 2 des VIC. Die Bits 2 bis 0 verschieben die Darstellung des linken Bildschirmrandes nach links. Hierdurch kann wie beim VDC ein weiches horizontales Scrollen um bis zu 7 Pixel erreicht werden.
Bit 3 schaltet den VIC von 39 Spalten (=0) auf 40 Spalten pro Zeile.
Gesetztes Bit 4 schaltet den Grafikschirm auf Multi-Color-Mode, in dem die Pixel doppelt breit dargestellt werden, dafür aber in mehr Farben als im Normalmodus.

Register 23 (Sprite-Vergrößerung horizontal)
Ein gesetztes Bit dieses Registers bewirkt, daß das Sprite derselben Nummer mit doppelter Breite dargestellt wird.

Register 24 (VIC-Speicheradressen)
Auch die einzelnen Speicher des VIC können verschoben werden. Dazu wählt man im Register 0 der später besprochenen CIA 2 einen 16 KByte langen Block innerhalb des Adreßraums von 64 K aus, in dem die Speicher des VIC liegen sollen. Die Speicherbank des Blocks wird im RAM-Configuration-Register der MMU gewählt. Die Bits 3 bis 1 des Registers 24 erlauben dann ein Verschieben des Zeichensatzes in Schritten von 2 KByte im Block. Das obere Halbbyte von Bit 7 bis 4 verschiebt das Video-RAM in 1-KByte-Schritten. Das Bit 3 ist außerdem gleichzeitig für die Verschiebung des Grafikspeichers zuständig, da Grafik und Zeichengenerator nicht zusammen gebraucht werden. Das Farb-RAM liegt unverschiebbar an der Adresse $d800.
Eine Besonderheit ist noch bei der Verschiebung des Zeichengenerators zu beachten: Werden der 16-K-Block und die Verschiebung so gewählt, daß der Zeichengenerator in den Bereich von $1000 bis $1fff oder $9000 bis $9fff fällt, wird durch die MMU immer das Zeichensatz-ROM angesprochen, das im Bereich von $d000 bis $dfff liegt. Der Grund hierfür ist einleuchtend, wenn man weiß, daß das Video-RAM normalerweise bei $400 in Bank 0 steht. Der Zeichensatz würde folglich mitten im BASIC-Programmtext stehen, wenn die Sonderregelung nicht getroffen worden wäre.

Register 25 (ICR)
Das Interrupt-Kontrollregister. In diesem Register wird festgehalten, welches die Ursache eines vom VIC ausgelösten Interrupts war:

Bit 0 = 1	Rasterzeilen-IRQ
Bit 1 = 1	Kollision Sprite mit Hintergrund
Bit 2 = 1	Kollision Sprite mit Sprite
Bit 3 = 1	Lichtgriffel
Bit 7 = 1	Eines oder mehrere der Bits 0 bis 3 sind gesetzt

Die Bits 4 bis 6 sind unbenutzt. Das Kontrollregister des VIC wird von den diesbezüglichen BASIC-Befehlen abgefragt. Nach Erkennen der Interruptursache muß das Register durch einfaches Beschreiben mit dem gelesenen Wert wieder gelöscht werden, damit der nächste Interrupt richtig erkannt werden kann.

Register 26 (IMR)
Im Interrupt-Maskenregister kann bestimmt werden, welches Ereignis der oben aufgeführten, wenn überhaupt eines, einen Interrupt auslösen soll. Die Belegung der Bits entspricht der des Kontrollregisters. Ein gesetztes Bit erlaubt den Interrupt.

Register 27 (Priorität)
Ein gesetztes Bit dieses Registers gibt dem Bildhintergrund die Priorität gegenüber dem zum Bit korrespondierenden Sprite. Das Sprite bewegt sich dann hinter Objekten auf dem Bildschirm. Ist ein Bit gelöscht, bewegt sich das entsprechende Sprite vor den Hintergrundzeichen und überdeckt sie.

Register 28 (Sprite-Multicolor)
Wieder ist jedem Bit ein Sprite zugeordnet. Wird das Bit gesetzt, so wird das Sprite im Multi-Color-Mode angezeigt.

Register 29 (Sprite-Vergrößerung vertikal)
Analog zu Register 23 verdoppelt sich die Länge eines Sprites durch das Setzen des zugehörigen Bits in diesem Register.

Register 30 (Sprite/Sprite-Kollision)
Kollision zweier Sprites. Wenn sich zwei Sprites berühren, werden die entsprechenden beiden Bits dieses Registers gesetzt. Im Kontrollregister 25 wird das Bit 2 gleich 1. Nach Auslesen des Registers 30 muß es wieder gelöscht werden.

Register 31 (Sprite/Hintergrund-Kollision)
Kollision eines Sprite mit einem Hintergrundzeichen. Es wird das zu dem Sprite gehörende Bit gesetzt sowie das Bit 1 des Kontrollregisters. Auch dieses Register muß nach dem Auslesen wieder gelöscht werden.

Register 32 (Rahmenfarbe)
Farbe des Rahmens. Die Aufschlüsselung der einzelnen Farbcodes entnehmen Sie bitte dem Handbuch.

Register 33 bis 36 (Hintergrundfarben)
Hintergrundfarben 0 bis 3. Im normalen Textmodus ist nur die Hintergrundfarbe 0 von Bedeutung.

Register 37+38 (Farben 0/1 Sprite-Multicolor)
Mögliche Farben der Sprites im Multicolormodus, je eine Farbe in einem Register. Im Multicolormodus angezeigte Sprites können insgesamt aus vier Farben bestehen, der Hintergrundfarbe, der Sprite-Farbe und den zwei Farben, die in diesen Registern festgelegt werden.

Register 39 bis 46 (Sprite-Farbe)
Hier kann eine individuelle Farbe eines Sprites gewählt werden. Zu Sprite 0 gehört das Register 39, zu Sprite 7 das Register 46.

Register 47 (Tastatur)
Ein neues Register gegenüber der VIC-Version des C 64. Das obere Halbbyte ist unbenutzt. Im unteren befindet sich die Information über den Zustand des Tastaturmatrix, Bits 3 bis 0. Mit diesem Register werden die gegenüber dem 64er hinzugekommenen Tasten abgefragt.

Register 48 (fast/slow)
Auch dieses Register ist neu. Es wird nur das Bit 0 benutzt. Ist es gesetzt, so wird der Systemtakt auf 2 MHz gesetzt, der VIC kann dann nicht mehr angesprochen werden.
Zusätzlich zu den beschriebenen VIC-Registern sind auch an anderer Stelle noch wichtige Speicherstellen des VIC vorhanden. Das Register 0 der CIA 2 wurde bei der Verschiebung der VIC-Speicher schon erwähnt.
Die 8 obersten Byte des jeweils eingestellten Video-RAMs gehören auch dazu. In ihnen wird je ein Zeiger auf die Definition eines Sprites aufbewahrt. Liegt das Video-RAM beispielsweise wie gewöhnlich bei $400, so enthält das Byte $7f8 den Zeiger für Sprite 0, das Byte $7ff den Zeiger für Nummer 7. Zur Angabe der Zeiger wird der 16-KByte-Adreßraum des VIC II in Blöcke zu je 64 Byte unterteilt, der Zeiger enthält dann die Nummer des 64-Byte-Blocks, in dem die Sprite-Definition zu finden ist.

1.3.2 Einzelne Modi des VIC II

Da der VIC II, was Sprites und Betriebsarten angeht, komplizierter aufgebaut ist als der VDC, muß hierzu noch ein Wort gesagt werden.
Beim VIC ist zunächst der Textmodus und der Grafikmodus zu unterscheiden. Beide Modi können wahlweise in der Normalfarben- oder Multicolorbetriebsart gefahren werden. Der Textmodus kennt zudem einen sogenannten Extended-Color-Mode.

1.3.2.1 Der Textmodus

Im Textmodus bezieht der VIC die Information zur Zeichendarstellung aus dem Video-RAM und dem Zeichensatz, der normalerweise aus dem Character-ROM bei $d000 gelesen wird. Jedes Byte des Video-RAMs enthält einen Zeiger auf die Position des darzustellenden Zeichens im Zeichensatz, den sogenannten Bildcode des Zeichens, der auch im Handbuch zum 128er zu finden ist. Im Zeichensatz steht die Zeichendefinition in Form eines 8-Byte-Blocks, dessen einzelne Bits den gesetzten oder nicht gesetzten Pixeln bei der Darstellung des Zeichens entsprechen.

In der Normalfarbenbetriebsart bestimmt das VIC-Register 33 die Farbe des Hintergrundes. Die Schriftfarbe wird für jedes einzelne Zeichen im sogenannten Farb-RAM ($d800) festgehalten, wobei jeweils das untere Halbbyte die Farbe gemäß der Aufschlüsselung des Handbuchs enthält.

Die Wirkung des Multicolormodus hängt davon ab, ob das Bit 3 des Farb-RAMs gesetzt ist. Ist dieses Bit gleich 0, so wird das zugehörige Zeichen wie im Normalmodus dargestellt, wobei 8 Farben zur Auswahl stehen, die mit den restlichen Bits des Farb-RAMs gewählt werden können. Ist Bit 3 gleich 1, wird das Zeichen im eigentlichen Multicolormodus angezeigt. Dazu werden je zwei Bits der Zeichendefinition zusammengefaßt als eine Farbinformation für zwei Pixel interpretiert. Die Farbe der beiden Pixel wird wie folgt festgelegt:

Bits	Farbe
0 0	Hintergrund 0 (Register 33)
0 1	Hintergrund 1 (Register 34)
1 0	Hintergrund 2 (Register 35)
1 1	die Farbe entstammt den restlichen drei Bits des Farb-RAMs

Der Extended-Color-Mode erlaubt eine Zeichendarstellung mit vier verschiedenen Hintergrundfarben. Die Schriftfarbe entstammt wie im Normalmodus dem Farb-RAM. Die zwei höchstwertigen Bits des Video-RAMs bestimmen die Hintergrundfarbe, so daß sich nur noch maximal 64 verschiedene Zeichen darstellen lassen. Die Zuordnung der Farben ist wie im Multicolormodus, nur die Kombination 1+1 spricht die Hintergrundfarbe 3 (Register 36) an.

1.3.2.2 Der Grafikmodus

Der Grafikmodus des VIC II wird erreicht, indem Bit 5 des Steuerregisters 1 (Registernummer 17) gesetzt wird. Bit 4 des Registers 22 schaltet die Multicolorbetriebsart ein und aus. Der HiRes-Schirm (High Resolution = Hohe Auflösung) hat eine Auflösung von 320 Pixeln in horizontaler und 200 Pixeln in vertikaler Richtung, sie ist damit halb so hoch

wie die des VDC, der allerdings, was die Farbe von Grafik anbelangt, nicht mit dem VIC mithalten kann. Als HiRes-Speicher werden 8000 KByte RAM benötigt. Jedes gesetzte Bit im Grafikspeicher entspricht im Normalmodus einem sichtbaren Pixel auf dem Bildschirm. Der Zusammenhang zwischen Zeilen- und Spaltenposition und der Position des zuständigen Bits im HiRes-Speicher ist etwas komplizierter als beim VDC. Die linke obere Ecke des Grafikschirms entspricht dem Bit 7 des ersten Byte des HiRes-Speichers. Die ersten 8 Byte definieren ein Achter-Päckchen von 8 Spalten und 8 Zeilen. Bit 7 des zweiten Byte ist somit der Pixel der zweiten Grafikzeile und ersten Grafikspalte, Bit 7 des dritten Byte der erste Pixel der dritten Zeile etc. In der ersten Zeile geht es weiter mit dem Byte Nummer 8, Bit 7 dieses Bytes ist dann der neunte Pixel in der ersten Zeile. So geht es weiter, bis 40 solcher 8-Byte-Päckchen voll sind. Dann folgen die nächsten 40 ab Zeile 9 des Grafikbildschirmes. Der Zusammenhang läßt sich auch leicht rechnerisch darstellen. Für eine beliebige Grafikzeile N ergibt sich die Startadresse S zu:

S = HiRes-Start + 8 * 40 * int(N/8) + N modulo 8
  = Start + 320 * int(N/8) + (N und 7)

In die Berechnung der Byte-Adresse A zu einer beliebigen Position (Zeile, Spalte) = (N,M) geht auch noch die Spalte mit ein:

A = S + 8 * int(M/8)

Die Rechnung der Bit-Position aus der Spalte hat sich gegenüber dem VDC natürlich nicht geändert, die Bytes sind ja nicht länger geworden:

Bit = 2 ↑ (7–(M und 7))

Die Farbinformationen zum Grafikschirm stammen aus dem Video-RAM, wobei ein Byte des Video-RAMs die Hintergrund- und die Pixelfarbe für ein Quadrat von 8*8 Pixeln festlegt. Die Hintergrundfarbe steht dabei im unteren Halbbyte, die Pixelfarbe im oberen. Das Farb-RAM hat in diesem Modus keine Funktion.
Wird Bit 4 des VIC-Registers 22 gesetzt, befindet man sich im Multicolor-Grafikmodus. Wie im Textmodus werden wieder 2 Bit des Grafikspeichers zusammengenommen und als Farbe für beide Pixel interpretiert. Die Zuordnung ist:

Bits	Farbe
0 0	Hintergrund 0
0 1	oberes Halbbyte des Video-RAMs
1 0	unteres Halbbyte
1 1	unteres Halbbyte des Farb-RAMs

1.3.2.3 Die Sprites

Sprites sind zunächst einmal frei bewegliche Grafikobjekte, die eine Größe von 24 Pixeln horizontal und 21 Pixeln vertikal besitzen. Sie können in einem Rahmen von 512*256 Rasterzeilen über den Bildschirm bewegt werden, also auch über die Bildschirmgrenzen hinaus. Jedes Sprite kann wahlweise horizontal und vertikal mit doppelter Ausdehnung dargestellt werden. Die Definition eines Sprites umfaßt 63 Byte, jedes gesetzte Bit in den Bytes entspricht in der Normaldarstellung einem sichtbaren Pixel in der Farbe des zugehörigen Sprite-Farbenregisters (Register 39 bis 46). Die Pixelfarbe ist die des Hintergrundes, wenn ein Bit nicht gesetzt ist.
Der Multicolormodus wird durch das Setzen eines Bits in Register 28 erreicht. Wie bei Text und Grafik werden die Bits der Sprite-Definition dann in Paaren als Farbinformation gelesen. Die Zuordnung der Farben ist:

Bits	Farben
0 0	durchsichtig
0 1	Farbe 0 (Register 37)
1 0	Farbe 1 (Register 38)
1 1	Sprite-Farbe (Register 39 bis 46)

1.3.3 Normale Speicherbelegung des VIC II

Das Video-RAM liegt im Textmodus an der Adresse $400 in Bank 0. Wird der Grafikschirm eingeschaltet, so belegt er den Bereich von $2000 bis $3fff, das Video-RAM wird an die Adresse $1c00 hochgelegt, damit die Farbinformationen des Grafikschirms nicht den Inhalt des 40-Zeichen-Textschirms überschreiben können.
Der Bereich von $e00 bis $fff ist für Sprite-Definitionen reserviert worden. $e00 entspricht einem Sprite-Definitionszeiger von 55 = 256 * 14/64 – 1.

1.4 Der Soundchip (SID, $d400)

Auch der Soundchip SID (Sound Interface Device) wurde vollkommen unverändert von C64 übernommen. Aufgrund der Leistungsfähigkeit des Chips wäre ein Wechsel auch nicht angebracht gewesen, zumal wohl jeder, der vom C64 kommt, mit der Bedienung des SID schon vertraut ist. Er erlaubt die Erzeugung dreier voneinander unabhängiger Stimmen, jede Stimme kann in ihrer Frequenz und Klangfarbe auf vielfältige Weise verändert werden. Die Stimmen können außerdem miteinander verknüpft werden, so daß ausgesprochen komplexe Klangbilder erzeugt werden können.

Des weiteren enthält der SID zwei Analog/Digital-Wandler, die einen extern am Computer angeschlossenen Widerstand messen und in digitaler Form aufbereiten können.

Die Register des SID

Register 0+1 (Frequenz Stimme 1)

Das Lowbyte der Frequenz steht in Register 0, das Highbyte in Register 1. Der Zusammenhang zwischen Frequenz in Hertz und Inhalt der Frequenzregister ist laut Handbuch ungefähr:

Frequenz/Hz	=17.029 * (Register Low + 256 * Register High)
oder	
Register	=0.0587 * Frequenz/Hz
Register High	=int (Register/256)
Register Low	=Register – 256 * Register High

Es ist allerdings zu erwarten, daß diese Richtwerte nicht exakt stimmen, da sie identisch mit denen des C64 sind, der eine geringfügig andere Taktfrequenz besitzt.

Register 2+3 (Pulsbreite Stimme 1)

Wird für die Stimme 1 als Schwingungsform die Rechteckspannung gewählt, so enthält dieses Registerpaar das Verhältnis von Dauer des Rechteckimpulses und Dauer der Pause zwischen zwei Rechtecken. Das obere Halbbyte des Registers 3 wird nicht benutzt, die Pulsbreite kann folglich zwischen 0 und 4095 liegen. Die genaue Mitte (2048) entspricht einem symmetrischen Verlauf des Rechtecksignals, bei dem die Pause zwischen zwei Impulsen und die Dauer des Impulses gleich ist.

Register 4 (Steuerregister Stimme 1)

Bit 0 (GATE) schaltet die Stimme 1 ein, sobald es gesetzt wird. Der Verlauf der Lautstärke eines angeschlagenen Tons gliedert sich in vier Phasen:

ATTACK	Die Lautstärke steigt innerhalb der in Register 5 einstellbaren Zeit auf den Maximalwert an (Register 24).
DECAY	Vom Maximalwert fällt der Lautstärkepegel in der ebenfalls in Register 5 einstellbaren Zeit wieder ab, bis die SUSTAIN-Lautstärke (Register 6) erreicht ist.
SUSTAIN	Die SUSTAIN-Lautstärke wird beibehalten, bis das GATE-Bit wieder Null wird.
RELEASE	Ist das GATE-Bit wieder Null, fällt die Lautstärke auf Null ab. Hierfür wird die in Register 6 eingestellte RELEASE-Zeit benötigt. Das Einleiten der RELEASE-Phase kann jederzeit durch das Rücksetzen des GATE-Bits erzwungen werden, auch während der ATTACK- oder DECAY-Phase.

Der ganze Vorgang ATTACK/ DECAY/ SUSTAIN/ RELEASE wird auch als Amplitudenmodulation bezeichnet. Der Verlauf der Lautstärke während der vier Phasen kann in Form einer Hüllkurve dargestellt werden.

Bit 1 (SYNC) fällt unter die Gruppe »special effects«. Durch das Setzen dieses Bits erreicht man, daß Stimme 1 mit Stimme 3 synchronisiert wird. Dies entspricht einem Phasenverlauf, in dem die Perioden beider Schwingungen starr aufeinander abgestimmt werden. In den Steuerregistern der Stimmen 2 und 3 erfolgt die Synchronisation jeweils mit Stimme 1 und 2.

Bit 2 (RING). Durch das Setzen dieses Bits erreicht man, daß das Dreieckssignal der Stimme 1 mit dem Signal der Stimme 3 multiplikativ verknüpft wird. Als Ergebnis erhält man eine komplexe Signalform. In den Steuerregistern der anderen Stimmen verknüpft das RING-Bit mit den Stimmen wie beim SYNC-Bit.

Bit 3 (RESET). Bei der Auswahl gleichzeitig mehrerer Schwingungsformen in den folgenden Bits kann es geschehen, daß der Rauschgenerator nicht anschwingt. Durch Setzen des Bits erhält der Generator einen Anstoß.

Bits 4 bis 7. Durch das Setzen eines oder mehrerer dieser Bits erfolgt die Auswahl der Schwingungsform(en) der Stimme 1. Mehrere Schwingungen werden logisch-UND verknüpft. Es bedeuten:

Bit 4 Dreiecksschwingung
Bit 5 Sägezahn
Bit 6 Rechteck
Bit 7 rosa Rauschen

Register 5 (ATTACK/ DECAY Stimme 1)

Im oberen Halbbyte wird die Dauer des Lautstärkeanstiegs bis zum Erreichen der Maximallautstärke eingestellt. Der kleinste Wert 0 entspricht einem Anstieg innerhalb von 2 msec, der größte Wert entspricht 8 sec.
Das untere Halbbyte gibt die Dauer des Lautstärkeabfalls vom Maximalwert bis auf den SUSTAIN-Pegel an. Der Einstellbereich reicht hier von 6 msec bis 24 sec.

Register 6 (SUSTAIN/ RELEASE Stimme 1)

Die Bits 3 bis 0 stellen den Lautstärkepegel dar, der während der SUSTAIN-Phase beibehalten wird. Ein Wert von 15 entspricht der Maximallautstärke.
Im oberen Teil des Registers wird die RELEASE-Zeit definiert, während der Ton vom SUSTAIN-Pegel auf Null abfällt. Die Zeitspanne kann von 6 msec bis 24 sec gewählt werden.

Register 7 bis 13 (Stimme 2)

Diese 7 Register entsprechen den Registern 0 bis 6 für die zweite Stimme.

Register 14 bis 20 (Stimme 3)

Genauso für die dritte Stimme.

Register 21+22 (Filterfrequenz)

Die Bits 7 bis 3 des Registers 22 sind unbenutzt. Die restliche 11 Bit lange Zahl legt die Grenz- beziehungsweise mittlere Frequenz der in den SID eingebauten Filter fest. Je nach Art des Filters bestimmt die Grenzfrequenz, welcher Teil des Frequenzspektrums durch den Filter gedämpft wird:

Hochpaß	Frequenzen unterhalb der Grenzfrequenz werden gedämpft
Tiefpaß	Frequenzen oberhalb der Grenzfrequenz werden gedämpft
Bandpaß	Die Frequenzen in einem Bereich um die Mittelfrequenz bleiben ungedämpft. Es wird also ein Teil des Frequenzspektrums bevorzugt.

Die Grenzfrequenz G errechnet sich zu:

$$G/Hz = 5.8 * R + 30$$

wenn R der Inhalt der Register 21/22 ist.

Register 23 (Filtergüte/ Filterquelle)

Die gesetzten Bits 0 bis 2 leiten jeweils die Stimme 1 bis 3 über den Filter, der aus dem eingebauten Hoch-, Tief- und Bandpaß zusammengesetzt wird.
Bit 3 leitet eine externe Schwingungsquelle über den Filter.
Die Bits 7 bis 4 bestimmen die Stärke der Resonanz des Filters. Frequenzen in der Umgebung der Resonanzfrequenz werden durch den Filter hervorgehoben, da dieser bei Erreichen der Resonanz mitschwingt. Wird die sogenannte Güte des Filters gleich Null gesetzt, ergibt sich keine Verstärkung der Frequenzen in der Nähe der Grenz- beziehungsweise Mittenfrequenz.

Register 24 (Lautstärke/ Filterauswahl)

Die unteren vier Bits stellen die Maximallautstärke für die drei Stimmen dar, die beim ATTACK angesteuert wird.
Bit 4 schaltet den Tiefpaß, Bit 5 den Bandpaß, Bit 6 den Hochpaß des Filters ein, wobei Kombinationen erlaubt sind.
Bit 7 macht die Stimme 3 unhörbar, was dann von Nutzen sein kann, wenn die dritte Stimme als Quelle für die Klangbilderzeugung bei den anderen beiden Stimmen benutzt wird.

WICHTIG! Alle bisher beschriebenen Register können ausschließlich beschrieben werden. Dafür können die folgenden Register nur gelesen werden:

Register 25+26 (A/D-Wandler 1 und 2)

An die Pins POTX und POTY der Controlports können Widerstände angeschlossen werden, die durch die beiden Analog/Digital-Wandler des SID gemessen werden können. Der Meßvorgang erfolgt nach dem Anschluß zyklisch und automatisch.

Register 27 (Rauschen Stimme 3)

Aus diesem Register kann der augenblickliche Stand des Generators der Stimme 3 ausgelesen werden, ein Zufallswert. Dazu muß dieser natürlich eingeschaltet sein.

Register 28 (Amplitude Stimme 3)

Die momentane Lautstärke der Stimme 3 kann man in diesem Register nachlesen, um etwa in Abhängigkeit dieses Wertes die Einstellungen der anderen Stimmen zu verändern.

1.5 Die CIAs (CIA1, $dc00, CIA2, $dd00)

Die CIAs (Complex Interface Adapters) übernehmen eine zentrale Rolle im C128. Sie sind zuständig für die serielle und parallele Ein- und Ausgabe sowie, damit eng verbunden, die Interrupt-Auslösung durch die in die CIAs eingebauten Timer und Echtzeituhren.
Die spezielle Rollenverteilung der beiden CIAs des 128ers wird später noch angesprochen. Sehen wir uns zuerst die Register eines CIA an:

1.5.1 Register des CIA

Register 0 (PRA)

Das Port-Register A dient der parallelen Ein-/Ausgabe. Jedes einzelne Bit entspricht einer Leitung, die durch das zum Port-Register gehörende sogenannte Daten-Richtungsregister zur Ein- oder Ausgabeleitung programmiert werden kann.
Wird das Port-Register ausgelesen, entspricht der gelesene Wert dem momentanen Zustand der an die entsprechenden Pins des Chips angeschlossenen Leitungen. Dies gilt sowohl für die Eingangs- als auch für die Ausgangsleitungen.

Register 1 (PRB)

Entspricht dem Port-Register A vollständig für 8 weitere Leitungen. Mit einem CIA können also 16 parallele Leitungen bedient werden (zum Beispiel 8 Eingangs- und 8 Ausgangsleitungen).

Register 2 (DDRA)

Das Daten-Richtungsregister zum Port-Register A. Die Abkürzung leitet sich aus dem Englischen her (Data Direction Register). Ein gesetztes Bit bestimmt die korrespondierende Leitung des Ports A zur Ausgangsleitung. Nicht gesetzte Bits dieses Registers legen entsprechend die Eingangsleitungen fest.

Register 3 (DDRB)

Dasselbe Register für das Port-Register B. Beide DDRs können gelesen und beschrieben werden.

Register 4+5/ 6+7 (Timer A/B)

Dies sind, wenn man so will, die beiden Stoppuhren des CIA. Die Zeitmessung erfolgt, indem der CIA von einem in die beiden Register des Timers geschriebenen Wert auf Null herunterzählt.
Der augenblickliche Zählerstand kann jederzeit ausgelesen werden. Das Register mit der niedrigeren Nummer enthält dabei immer das Lowbyte des Zählerstandes.
Das Beschreiben der Register erfolgt nicht direkt, vielmehr werden die in die Register geschriebenen Daten zunächst in einen Zwischenspeicher gebracht, von dem aus sie durch das jeweilige Steuerregister in den Timer geladen werden.
In den Steuerregistern (CRA und CRB) können außerdem die verschiedenen Betriebsarten der Timer ausgewählt werden.

Register 8 (TOD 10ths)

Der CIA besitzt eine eingebaute 24-Stunden-Echtzeituhr mit einer kleinsten Anzeige von $^1/_{10}$ sec. Zur Zeitmessung dient die Netzfrequenz, wobei dem CIA in Register 14 (CRA) mitgeteilt werden muß, ob es sich um eine Frequenz von 50 oder 60 Hz handelt.
An der Echtzeituhr kann auch eine Alarmzeit eingestellt werden, zu der ein Interrupt ausgelöst werden kann.
Das vorliegende Register enthält die Zehntelsekunden der Echtzeituhr, wie bei allen Registern der Uhr, im BCD-Format.
Das BCD-Format ist dadurch gekennzeichnet, daß ein Byte nur zur Darstellung der Zahlen von 0 bis 99 benutzt wird. Zu diesem Zweck unterteilt man das Byte in die beiden Halbbytes, die jeweils eine Ziffer enthalten. Die Ziffern selbst werden wie gewohnt binär dargestellt. Beispiel:

```
12   =% 0001 0010
58   =% 0101 1000
99   =% 1001 1001
```

Das Auslesen des Registers 8 ergibt die Zehntelsekunden der Echtzeituhr. Wird das Register beschrieben, entscheidet das Bit 7 des Kontrollregisters B, ob die Uhr gestellt wird (Bit 7 = 0) oder ob die Alarmzeit eingegeben wird (Bit 7 = 1).

Die oberen 4 Bit des Registers sind immer Null. Es muß darauf geachtet werden, daß dies auch beim Beschreiben von TOD 10ths eingehalten wird.

Register 9/ 10 (TOD sec/ TOD min)

Sekunden und Minuten der Uhr. Da nur bis 60 gezählt werden muß, ist das Bit 7 der beiden Register immer gleich Null. Ansonsten gilt das für das Register TOD 10ths Gesagte.

Register 11 (TOD h)

Das Stundenregister der Uhr. Wichtig ist, daß die Stunden nicht bis 24 gezählt werden, sondern wie in den englischsprachigen Ländern üblich bis 12, wobei die Angabe AM (vormittags) beziehungsweise PM (nachmittags) hinzukommt. Bit 7 des Registers ist diese Angabe, Bit 7 = 0 entspricht AM. Sonstiges wie bei Register 8.

Register 12 (SDR)

Das serielle Datenregister. Das Register kann ausgelesen und beschrieben werden.
Im CRA (Register 14) kann festgelegt werden, ob SDR als Eingabe- oder Ausgaberegister arbeitet. Die Arbeitsweise ist so, daß, je nach Betriebsart, entweder die im SDR enthaltenen Daten bitweise aus dem Register heraus auf eine an den CIA angeschlossene serielle Leitung geschoben werden (Bit 0 voran), oder die Daten, die eine serielle Leitung übergibt, bitweise in das SDR hineingeschoben werden.
Zusätzlich ist zu dieser Art der Datenübertragung ein Taktsignal notwendig, das das Schieben des Registers steuert.
Im Eingabemodus stammt dieses Taktsignal von einer externen Quelle und wird auf einer zweiten Leitung neben der Datenleitung (DATA) empfangen.
Im Ausgabemodus übernimmt Timer A die Erzeugung des Taktes. Der erzeugte Takt erscheint gleichzeitig auf der Taktleitung (CLOCK) neben der Datenleitung. Die Taktfrequenz beträgt die halbe Zählfrequenz des Timers A. Timer A wird zur Erzeugung der Taktfrequenz in einem Modus betrieben, in dem er periodisch von einem konstanten Wert auf Null herabzählt und bei Erreichen der Null von vorn beginnt (CONTINUOUS).

Register 13 (ICR)

Das sogenannte Interrupt-Kontrollregister. Dieses Register dient gleichzeitig dazu, die Ursache eines Interrupts festzustellen (Lesezugriff), wie dazu, das Eintreten eines Ereignisses (Alarmzeit, Timer-Unterlauf) mit einem Interrupt zu verbinden (Schreibzugriff, Eingabe der Interruptmaske).
Die Zuordnung der einzelnen Bits des ICR in der Interruptmaske ist:

Bit 0 Timer A Unterlauf
Bit 1 Timer B Unterlauf
Bit 2 Uhrzeit = Alarmzeit
Bit 3 Serielles Datenregister voll oder leer (je nach Betriebsart)

Bit 4 Signal am FLAG-Pin des CIA
Bit 5 immer Null
Bit 6 immer Null

Mit Bit 7 hat es eine besondere Bewandtnis: Ist Bit 7 beim Beschreiben des ICR gleich 1, so setzt jedes geschriebene 1-Bit das entsprechende Bit des Registers. Ist es 0, so löscht jedes gesetzte Bit das entsprechende Bit des Registers.
Nach Setzen des ICR erlaubt jedes gesetzte Bit des ICR den Interrupt durch das dem Bit zugeordnete Ereignis. Ist Bit 1 z.B. gleich 1, führt jeder Unterlauf von Timer B zum Interrupt, ist Bit 2 gleich 1, wird ein Interrupt durch den Alarm der Echtzeituhr ausgelöst. Natürlich sind Kombinationen erlaubt.
Das Auslesen des ICR ermöglicht dem Prozessor, die Ursache eines Interrupt herauszufinden und entsprechend zu reagieren. Jedes beim Lesen des Registers gesetzte Bit entspricht dem korrespondierenden Ereignis. Bit 7 ist gesetzt, wenn mindestens eines der anderen Bits gesetzt ist. Durch das Lesen des Registers wird es gleichzeitig gelöscht, es kann also nur einmal sinnvoll gelesen werden!

Register 14 (CRA)

Kontrollregister zu Timer A. In den beiden höchstwertigen Bits werden außerdem noch Einstellungen der Uhr und des seriellen Ports vorgenommen:
Bit 7: Ist das Bit gleich Null, bedeutet dies, daß die Netzfrequenz 60 Hz beträgt, sonst ist sie 50 Hz. Hierauf sollte man achten, wenn die Zeitanzeige korrekt ablaufen soll.
Gesetztes Bit 6 schaltet den seriellen Port auf Ausgang. Bit 6 = 0 bedeutet dementsprechend, daß Daten seriell empfangen werden können.
Bit 5: Ist das Bit gleich 0, so zählt der Timer A mit der Geschwindigkeit des Systemtaktes. Ist es gesetzt, zählt der Timer im Takt eines an den CNT-Pin angelegten externen Signals.
Bit 4 dient dem Laden des Timers A mit einem Startwert. Dabei ist es gleichgültig, ob der Timer gerade läuft oder nicht. Das Bit wird gesetzt, wenn das Laden erfolgen soll.
Bit 3 bestimmt die Betriebsart des Timers A. Es kann gewählt werden zwischen dem ONE-SHOT-Modus, in dem der Timer einmal vom Startwert auf Null zählt und dann anhält (Bit 3 = 1), oder dem CONTINUOUS-Modus, in dem sich der Zählvorgang periodisch wiederholt (Bit 3 = 0).
Bit 1 bestimmt, ob das von Timer A beim Unterlauf erzeugte Signal über den Pin PB6 auf den parallelen Ausgang gelegt wird. Durch das Setzen des Bits wird dies veranlaßt, wobei es ohne Bedeutung ist, ob im Daten-Richtungsregister B die siebte Leitung (= Bit 6) als Eingang oder als Ausgang festgelegt wurde.
Bit 2 bestimmt die Form des durch Bit 1 erzeugten Signals. Ist das Bit gelöscht, erscheint der Unterlauf an PB6 in der Form eines Impulses mit der Dauer eines Systemtaktes. Ist Bit 2 = 1, ändert sich der Zustand der Leitung PB6 mit jedem Unterlauf. Das heißt, einmal angenommen, die Leitung sei ursprünglich auf Pegel 0, so daß der erste Unterlauf des

Timers A die Leitung auf 1-Pegel bringt, der zweite Unterlauf bringt PB6 wieder auf Pegel 0 usw.
Bit 0 startet den Timer A (Bit 0 = 1) beziehungsweise stoppt ihn (Bit 0 = 0).

Register 15 (CRB)

In den unteren 5 Bits werden dieselben Aufgaben wie im Kontrollregister A erfüllt, nur bezogen auf den zweiten Timer. Wird das Unterlaufsignal auf den Parallelausgang gelegt, so geschieht dies über den Pin PB7 (Bit 7 PRB).
Die Bits 6 und 5 erlauben die Wahl, was in Timer B gezählt werden soll. Es gilt:

Bit 6	Bit 5	zu zählen:
0	0	Systemtakte
0	1	externes Signal über CNT
1	0	Unterläufe von Timer A
1	1	Unterläufe von Timer A, falls gleichzeitig CNT gleich 1 ist

In Bit 7 wird festgelegt, ob ein Schreibzugriff auf die Register der Echtzeituhr zum Setzen der Uhrzeit oder zum Setzen der Alarmzeit erfolgt. Bit 7 = 0 entspricht dem Setzen der Uhrzeit.

1.5.2 Die Rolle der CIAs im 128er

1.5.2.1 Der CIA 1 ($dc00)

Der CIA 1, der sogenannte IRQ-CIA, leistet im wesentlichen folgendes im 128er:

- PRA und PRB sind normalerweise mit der Tastatur verbunden. Hier geschieht also die Tastaturabfrage. Timer A liefert den regelmäßigen, etwa 60mal pro Sekunde erfolgenden Interrupt zur Tastaturabfrage.
- Bei Benutzung von Joysticks merkt sich der Basic-Interpreter die Inhalte von DDRA und DDRB und benutzt PRA bzw. PRB als Joystick-Eingang. Die Bedeutung der einzelnen Bits der Ports ist in diesem Fall:

Bit 0	oben
Bit 1	unten
Bit 2	rechts
Bit 3	links
Bit 4	Feuerknopf

- Die CNT- und die serielle DATA-Leitung des CIA 1 sind mit dem Userport verbunden (Anschlüsse 4 und 5).
- Timer A wird bei Diskettenbetrieb und Timer B bei Kassettenbetrieb benötigt.
- Timer A und der serielle Port (SP) des CIA 1 sind zuständig für die schnelle serielle Datenübertragung.

1.5.2.2 Der CIA 2 ($dd00)

Der CIA 2 betreibt die notwendigen Ein- und Ausgänge, die zum seriellen IEC-Bus und zum Userport führen. Das Port-Register A (Registernummer 0) enthält:
- In den Bits 1 und 0 die höchstwertigen Bits des VIC-II-Speichers.
- Bit 2 wird in Verbindung mit einer RS232-Cartridge genutzt. Die entsprechende Leitung befindet sich am Anschluß M des Userports.
- Bit 3 ist der Attention-Ausgang (ATN) des seriellen Bus. Er ist auch bei Pin 9 am Userport wiederzufinden. Diese Leitung dient dazu, alle am seriellen Bus angeschlossenen Geräte darauf vorzubereiten, daß eines von ihnen gleich angesprochen werden wird. Welches dann tatsächlich angesprochen ist, ergibt sich erst aus der nachfolgenden Sendung der Geräteadresse.
- Bit 4 ist der CLOCK-Ausgang des IEC-Bus. Eine serielle Datenübertragung ist ja nur dann sinnvoll, wenn zugleich mit den Daten auch der Takt übertragen wird, mit dem die Daten gesendet werden.
- Bit 5 ist schließlich die Ausgangsleitung für die Daten, die der Prozessor sendet.
- Die Bits 6 und 7 stellen die gleichen Leitungen als Eingang vom IEC-Bus dar.

PRB des CIA 2 ist im Normalbetrieb mit den Anschlüssen C bis L des Userports verbunden. Wird ein RS232-Zusatz verwendet, befinden sich in PRB RS232-Leitungen.
Die serielle Leitung und die CNT-Leitung des CIA 2 belegen am Userport die Anschlüsse 6 und 7.
Die beiden Timer des CIA 2 sind solange unbenutzt, wie keine RS232-Übertragung vorgenommen werden soll. Ist dies doch der Fall, dienen sie zur Festlegung der Übertragungsgeschwindigkeit, auch Baudrate genannt (Baud = Bits/sec).

2
Die Routinen der Common Area (0 – $3ff)

Thema: Beschreibung der Kernal- und BASIC-Unterprogramme im gemeinsamen RAM-Bereich von 0 bis $3ff.

Wie schon bei der Betrachtung der Speicherverwaltung (MMU) festgestellt, können Daten zwischen verschiedenen Speicherbänken nur über die Common Area ausgetauscht werden, sieht man einmal vom Einsatz des RAM-Konfigurationsregisters ab, mit dem unabhängig von der Common Area zumindest die Zeropage und der Stack-Bereich zeitweilig in andere Bänke verlegt werden können.
Im Prinzip gibt es nur zwei Arten, die Common Area zu nutzen, die beide von BASIC und Betriebssystem angewendet werden:
Entweder stehen die Daten im gemeinsamen Bereich, wo sie aus jeder Bank heraus sichtbar sind, siehe Zeropage oder Stack im Normalfall, oder es stehen die Programme beziehungsweise Unterprogramme dort, die Daten aus anderen Bänken holen.
In der normalerweise eingestellten Common Area, die den Bereich von 0 bis $3ff umfaßt, also bis zum Beginn des Video-RAM des VIC II reicht, liegen zwei Arten solcher Unterprogramme:

2.1 Kernal-Routinen in der Common Area

Insgesamt fünf Routinen stellt das Betriebssystem dem Maschinensprache-Programmierer im gemeinsamen RAM-Bereich zur Verfügung. Beim Einschalten oder bei einem Reset werden diese aus dem Kernal-ROM in die Common Area kopiert.

Diese Routinen werden von BASIC 7.0 so gut wie nicht benutzt, da BASIC eigene Routinen in der Common Area besitzt – Ausnahmen sind die weiten Sprünge in den Befehlen SYS und BOOT sowie die Monitorbefehle. Die Kernal-Routinen der Common Area sind damit praktisch ohne Einschränkungen für eigene Zwecke nutzbar, man beeinflußt durch ihren Gebrauch weder den Interpreter noch das Betriebssystem.

2.1.1 Laden, Speichern, Vergleichen (FETCH, $2a2, STASH, $2af, CMPARE, $2be)

Die FETCH-Routine im Listing:

```
FETCH = $2a2

Adresse     Befehle

02a2        lda $ff00       altes CR in den Akku
02a5        stx $ff00       CR mit X-Register laden
02a8        tax             altes CR in X sichern
02a9        lda ($ff),y     indirekt nachindiziertes Laden des Akkus
02ab        stx $ff00       altes CR wiederherstellen
02ae        rts             Rückkehr zum aufrufenden Programm
```

Die Arbeitsweise und die Anwendung der FETCH-Routine sind recht einfach zu verstehen:
Der erste Schritt vor Benutzung von FETCH besteht darin, den in der Speicherzelle

FETVEC = $2aa

veränderlichen Pointer auszuwählen, über den indirekt nachindiziert geladen werden soll. Dies kann ohne weiteres geschehen, denn es handelt sich ja nicht um eine unveränderliche ROM-Routine, sondern um RAM-Speicherstellen. Zur Erinnerung: Indirekt nachindiziertes Laden über einen Pointer der Zeropage läuft wie folgt ab. Der Zeropage-Pointer, bestehend aus 2 Byte, enthält eine Adresse in der Folge Low-/Highbyte. Zu dieser Adresse addiert der Prozessor für sich selbst, also ohne Veränderung des Pointers, den Inhalt des Y-Registers, um die Adresse zu erhalten, aus der geladen werden soll.

Als zweites erwartet FETCH im X-Register die Konfiguration, in der sich die besagte Adresse befindet. Im X-Register wird also die RAM-Bank gewählt und bestimmt, welche ROMs ein- beziehungsweise ausgeschaltet sind (siehe Konfigurationsregister der MMU).

Hieraus sieht man, daß die FETCH-Routine nicht nur dazu benutzt werden kann, andere RAM-Bänke anzusprechen, sondern auch innerhalb einer Bank, etwa beim Laden von Adressen unter sonst eingeschalteten ROMs, von Nutzen ist.

Nach Ablauf der Routine ist der alte Wert des Konfigurationsregisters wiederhergestellt, der Rücksprung zum aufrufenden Programm ist also gewährleistet.

Zur Erleichterung der Arbeit mit den Kernal-Routinen, die sich auf andere Konfigurationen beziehen, wurde von Commodore der sogenannte Konfigurationsindex eingeführt, den man tunlichst nicht mit der Konfiguration, dem Inhalt des Konfigurationsregisters, verwechseln sollte. Der Index weist in eine Tabelle, die im Kernal bei Adresse $f7f0 zu finden ist. Jedem Index ist in der Tabelle ein bestimmter Konfigurationswert zugeordnet:

Index	Konfiguration	Erläuterung
0	% 0011 1111	Bank 0 durchgehend RAM
1	% 0111 1111	Bank 1 nur RAM
2	% 1011 1111	Bank 2 nur RAM (entspr. Index 0)
3	% 1111 1111	Bank 3 nur RAM (normalerweise wie 1)
4	% 0001 0110	Bank 0, 1, 2 beziehungsweise 3, I/O sichtbar,
5	% 0101 0110	internes Zusatz-ROM in den Bereichen von
6	% 1001 0110	$8000 – $bfff und $c000 bis $ffff
7	% 1101 0110	
8	% 0010 1010	wie die Indizes 4 bis 7, nur daß in den
9	% 0110 1010	ROM-Bereichen das externe Zusatz-ROM
10	% 1010 1010	angesprochen wird
11	% 1110 1010	
12	% 0000 0110	Bank 0 mit Kernal-ROM und I/O. Dazu der untere Teil des internen Zusatz-ROMs von $8000 bis $bfff
13	% 0000 1010	dasselbe für das externe Zusatz-ROM
14	% 0000 0001	Kernal- und beide BASIC-ROMs von $4000 bis $7fff und $8000 bis $bfff. Im Bereich von $d000 bis $dfff ist der Zeichensatz sichtbar
15	% 0000 0000	dasselbe wie unter Index 14, nur ist der Bereich von $d000 bis $dfff von der I/O belegt (Register von VIC, VDC, ... und Farb-RAM des VIC)

Die Benutzung des Konfigurationsindexes anstatt der eigentlichen Konfiguration hat natürlich den Vorteil, daß man sich nicht jedesmal relativ kompliziert zu überlegen hat, wie das Konfigurationsregister aussehen muß, wenn man eine bestimmte Speicherbelegung erreichen will. Es lassen sich aber ebenso natürlich nicht alle möglichen

Konfigurationen und auch nicht alle benötigten Konfigurationen mit Hilfe der Indizes ermitteln. So wird man eine Konfiguration, in der nur das Kernal-ROM eingeschaltet ist (% xx00 1110), vergeblich suchen.

Die Verknüpfung zwischen Konfigurationsindex und dem zugehörigen Konfigurationswert übernimmt die Kernal-Routine GETCFG ($ff6b). Sie liefert zu einem im X-Register übergebenen Index im Akkumulator die aus der Tabelle entnommene Konfiguration.

Zu jeder der Kernal-Routinen in der Common Area existiert ein Gegenstück im Kernal-ROM, das mit Konfigurationsindizes arbeitet und gleichzeitig die Abspeicherung des Pointers in den FETVEC oder die entsprechenden Speicherzellen der anderen Routinen übernimmt.

Im vorliegenden Fall liegt diese Routine, die zur Unterscheidung INDFET genannt wird, bei $ff74.

INDFET = $ff74

Der Aufruf dieser Routine erfolgt mit den Übergabewerten:

Akku	Lowbyte des Pointers, über den geladen werden soll
X	Index der Konfiguration, unter der die Ladeadresse aufzufinden ist
Y	bleibt unbeeinflußt und enthält den Offset von der Ladeadresse

Möchte ich zum Beispiel über den Pointer $fa einen Wert aus der Speicherbank 1 laden, wobei kein ROM eingeschaltet werden soll, sieht dies mit den beiden Routinen so aus:

```
lda #$fa     Pointer in den FETVEC bringen
sta $02aa
ldx #$7f     CR für Bank 1 ohne ROMs
jsr $02a2    FETCH

lda #$fa     Pointer
ldx #1       Konfigurationsindex
jsr $ff74    INDFET
```

Die Unterprogramme zum Abspeichern (STASH) und Vergleichen (CMPARE) sehen ganz analog zu FETCH aus:

STASH = $2af

```
02af     pha           abzuspeicherndes Byte merken
02b0     lda $ff00     altes CR merken
02b3     stx $ff00     neues CR einsetzen
02b6     tax           altes CR nach X retten
02b7     pla           abzuspeicherndes Byte vom Stack holen
02b8     sta ($ff),y   und speichern
02ba     stx $ff00     altes CR wiederherstellen
02bd     rts           Rücksprung
```

CMPARE = $2be

```
02be     pha           Vergleichswert auf den Stack
02bf     lda $ff00     altes CR merken
02c2     stx $ff00     neues CR setzen
02c5     tax           altes CR nach X retten
02c6     pla           Vergleichswert vom Stack holen
02c7     cmp ($ff),y   und den Vergleich durchführen
02c9     stx $ff00     altes CR wieder einsetzen
02c      rts           Ende
```

Die Vektoren, in denen die Pointer festgelegt werden, über die abgespeichert beziehungsweise verglichen wird, sind:

STAVEC = $2b9

CMPVEC = $2c8

Auch zu STASH und CMPARE gibt es ROM-Varianten wie zu FETCH. Es sind dies:

INDSTA = $ff77

INDCMP = $ff7a

Die Bedienung dieser ROM-Routinen erfolgt genauso wie bei INDFET.

Ganz interessant ist vielleicht die Möglichkeit, die STASH- und die CMPARE-Routine so umzubauen, daß sie andere Operationen als in der ursprünglichen Form ausführen. Hierzu ist es lediglich notwendig, den Befehlscode der gewünschten Operation nach STAVEC-1 beziehungsweise CMPVEC-1 zu schreiben. So wäre es zum Beispiel denkbar, aus CMPARE ein ADDFAR zu machen:

```
lda #$71          Opcode für adc (...),y
sta cmpvec-1
lda #pointer
sta cmpvec
clc
ldx #conf         Konfiguration
jsr cmpare
```

Dabei darf man selbstverständlich nicht vergessen, die CMPARE-Routine nach Gebrauch wieder zu restaurieren.

Die möglichen Opcodes, die für eine Änderung von STASH und/oder CMPARE gebraucht werden können, sind:

```
adc  $71
and  $31
cmp  $d1
eor  $51
ora  $11
sbc  $f1
sta  $91
```

Eine wichtige Frage müßte sich dem geübteren Programmierer in Maschinensprache im Verlauf der Beschreibung der Lade-, Speicher- und Vergleichsroutine eigentlich schon gestellt haben:

Wo ist der Befehl »sei«, der verhindert, daß der Prozessor beim Auftreten eines Interrupts in den Tiefen der Vogesen verschwindet, wenn das Kernal-ROM abgeschaltet wird?

Immerhin 60mal in der Sekunde erfolgt ja der Tastatur-Interrupt!

Dies kann man leicht klären, wenn man den eingebauten Monitor des 128ers einmal einschaltet und die Bereiche oberhalb von $ff00 in den RAM-Bänken disassembliert:

Beim Einschalten des Computers beziehungsweise bei einem Reset werden nämlich alle für die Behandlung von Interrupts notwendigen Routinen in alle vorhandenen RAM-Bänke kopiert! Dies geschieht gleichzeitig mit dem Kopieren der Routinen der Common Area in der Routine COPY1 ($e0cd). Der Bereich oberhalb von $ff00 ist aus diesem Grund erst einmal tabu für eigene Programme – es sei denn, man will auf eine exotische Art den Interrupt oder den Reset beeinflussen, wenn das Kernal-ROM ausgeschaltet ist.

2.1.2 Die »weiten« Sprünge (JMPFAR, $2e3, JSRFAR, $2cd)

Die zweite Art von Kernal-Routinen in der Common Area ermöglicht den Aufruf von Programmen, die innerhalb anderer Konfigurationen stehen. Der Fall, daß sie in anderen RAM-Bänken stehen, ist hier mit enthalten, es muß aber nicht notwendig der Fall sein.

Im Zusammenhang mit JMPFAR und JSRFAR gewinnen die Zeropage-Adressen 2 bis 9 ihre Bedeutung. In ihnen werden vor dem Aufruf alle Parameter abgelegt, die beim Aufruf einer Routine durch JMPFAR oder JSRFAR benötigt werden, wie die Adresse des aufzurufenden Programms, in welcher Konfiguration es steht, welchen Inhalt die einzelnen Prozessorregister beim Aufruf enthalten sollen etc.
Im Fall von JSRFAR werden an diesen Adressen auch die entsprechenden Werte bei der Rückkehr vom aufgerufenen Programm nach JSRFAR zurückgeliefert.

Die Zuordnung der einzelnen Speicherstellen zu den Registern usw. ist:

Adresse	Name	Bedeutung
2	bank	Index (!) der Konfiguration, in der das angezielte Programm steht
3	pchi	Highbyte der Programmadresse
4	pclo	Lowbyte der Programmadresse
5	sreg	Statusregister
6	areg	Akkumulator
7	xreg	X-Register
8	yreg	Y-Register
9	stkptr	Stackpointer

Sehen wir uns jetzt einmal das Listing der JMPFAR-Routine an, damit wir erkennen, welche dieser Bytes die Routine an das aufgerufene Programm übergibt:

JMPFAR ($2e3)

```
02e3    ldx #0    Der Programmzähler und das Statusbyte
02e5    lda 3,x   werden auf den Stack gelegt. Der Bef.
02e7    pha       rti bewirkt, daß das Statusbyte vom
02e8    inx       Stack ins Statusregister gelangt, und
02e9    cpx #3    daß der Programmzähler auf den Wert d.
02eb    bcc $02e5 nächsten beiden Bytes auf dem Stack gesetzt wird
02ed    ldx 2     Index der Ziel-Konfiguration
02ef    jsr $ff6b GETCFG: zugehöriges CR aus der Tabelle
02f2    sta $ff00 CR mit dem Wert setzen
```

```
02f5    lda 6     Akku,
02f7    ldx 7     X-Register,
02f9    ldy 8     Y-Register laden
02fb    rti       und Sprung
```

Offenbar werden alle Parameter außer stkptr an das aufgerufene Programm weitergegeben.
Wir erkennen aus dem Listing weiter, daß das Kernal-ROM beim Aufruf von JMPFAR eingeschaltet sein muß, denn von JMPFAR wird die Kernal-Routine GETCFG benötigt. Dies ist zumindest in der Grundversion von JMPFAR so, denn ebenso wie FETCH oder STASH kann auch JMPFAR natürlich von uns verändert werden, wenn man einen Punkt dabei berücksichtigt:
JMPFAR steht nicht uneingeschränkt nur für eigene Anwendungen zur Verfügung. Der Monitorbefehl G zum Beispiel oder der BASIC-Befehl SYS machen ebenfalls von JMPFAR Gebrauch. Ändern wir also JMPFAR, müssen wir dafür sorgen, daß die Änderung auch rechtzeitig wieder rückgängig gemacht wird, bevor die Routine von BASIC oder dem Monitor benutzt wird.
Die einzige Änderung, die man höchstwahrscheinlich an JMPFAR planen könnte, ist die Aufhebung der Regel, daß das Kernal-ROM eingeschaltet sein muß, wenn JMPFAR aufgerufen wird. Dies kann man auf zweierlei Art erreichen:
Als erstes könnte man den Befehl jsr GETCFG in drei NOPs umwandeln, der Aufruf einer Kernal-Routine würde damit entfallen. Zusätzlich müßte nur noch der vorhergehende Befehl ldx bank durch lda bank ersetzt werden, und man hat eine JMPFAR-Routine, die nicht mit dem Konfigurationsindex, sondern mit der Konfiguration selbst arbeitet.
Die zweite Möglichkeit läßt die eigentliche JMPFAR-Routine unverändert. Wir gehen hier einfach so vor, daß wir die GETCFG-Routine samt Tabelle der Konfigurationen an die Adresse $ff6b in alle RAM-Bänke kopieren, genau wie es das Betriebssystem mit den IRQ-Routinen macht. Lediglich die absolute Tabellenadresse in GETCFG muß bei der Übertragung angepaßt werden. Der entsprechende Platz von 20 Byte ist frei, wie man sich mit dem Monitor überzeugen kann. Das folgende Programm könnte das Kopieren übernehmen:

```
0b00    lda #$ff       Highbyte der Zieladresse
0b02    sta $fe        in das Highbyte des Pointers
0b04    lda #$6b       Lowbyte der Zieladresse
0b06    sta $fd        in das Lowbyte des Pointers
0b08    lda #$fd       Pointer soll $fd/$fe sein
0b0a    sta $2b9       STAVEC
0b0d    ldy #19        Insgesamt 20 Byte zu kopieren
0b0f    ldx #$3f       CR für Bank 0 alles RAM
0b11    lda $f7ec,y    $f7ec ist die eigentliche Adresse von GETCFG
```

```
0b14    jsr $2af      STASH
0b17    dey           Schleifen, bis
0b18    bpl $0b0f     negativ = 20 Byte übertragen
0b1a    ldy #1        Tabellenadresse anpassen
0b1c    lda #$6f      neues Lowbyte der Adresse
0b1e    ldx #$3f      in Bank0 speichern
0b20    jsr $2af
0b23    iny           und neues Highbyte
0b24    lda #$ff
0b26    ldx #$3f      abspeichern
0b28    jsr $2af
0b2b    ldy #19       dasselbe noch einmal für die
0b2d    ldx #$7f      Übertragung in die Bank 1
0b2f    lda $f7ec,y
0b32    jsr $2af
0b35    dey
0b36    bpl $0b2d
0b38    lda #$6f
0b3a    ldy #1        und wieder die Tabellenadresse
0b3c    ldx #$7f      anpassen
0b3e    jsr $2af
0b40    iny
0b41    ldx #$7f
0b43    lda #$ff
0b45    jsr $2af
0b48    rts
```

Daß JMPFAR nach der Übertragung von GETCFG in die RAM-Bänke funktioniert, auch wenn das Kernal-ROM ausgeschaltet ist, wird nach der Erläuterung von JSRFAR noch an einem Beispiel gezeigt.
Ich glaube, man sieht auch recht deutlich an dem Kopierprogramm, warum Commodore die Konfigurationsindizes eingeführt hat: Das ständige Laden des X-Registers mit der Konfiguration kostet doch eine Menge unnötigen Speicherplatz, den man besser nutzen kann.

Die JSRFAR-Routine ist das Gegenstück zu JMPFAR, wenn ein Unterprogramm aus einer anderen Konfiguration aufgerufen werden soll – zum Beispiel eine BASIC-Routine von einer Adresse aus, die unter dem BASIC-ROM liegt, oder auch von Bank 1 aus eine Kernal-Routine, wobei die Bank 0 eingeschaltet sein soll, oder was man sich sonst vorstellen kann.
Das Listing der JSRFAR-Routine:

JSRFAR ($2cd)

```
02cd    jsr $2e3     JMPFAR wird aufgerufen
02d0    sta 6        die Übergabeparameter werden in der
02d2    stx 7        Zeropage
02d4    sty 8        abgelegt
02d6    php          Status über den Stack
02d7    pla          in den Akkumulator
02d8    sta 5        und nach sreg
02da    tsx          Stapelzeiger
02db    stx 9        nach stkptr
02dd    lda #0       JSRCFG = $2de: altes CR wieder-
02df    sta $ff00    herstellen
02e2    rts          und zurück
```

Zurückgeliefert werden von JSRFAR nach dem Aufruf sämtliche Register, der Stapelzeiger und das Statusbyte.

Vor Benutzung von JSRFAR muß auf jeden Fall die ursprüngliche Konfiguration in

JSRCFG = $2de

abgespeichert werden, sonst ist eine Rückkehr zum aufrufenden Programm nicht möglich!

Sehen wir uns jetzt einmal das versprochene Beispiel an. Es soll darin bestehen, daß von der Adresse $c000 unter dem Kernal-ROM in Bank 1 ein kleiner Text, bestehend aus dem Buchstaben »a«, ausgegeben wird. Dies geschieht mit der Kernal-Routine, die sich BSOUT nennt. BSOUT wird später noch ausführlicher besprochen. Für den Augenblick reicht es, zu wissen, daß diese Routine in der Lage ist, ein Zeichen, das im Akku übergeben wird, auf den Bildschirm auszugeben. Um das Beispiel wie geplant ablaufen zu lassen, ist es notwendig, zuerst das vorher stehende Kopierprogramm zu starten. Wer sich dies ersparen will, kann das Kopieren auch mit dem Monitor vornehmen. Hierzu sind die vier Befehle

```
t ff7ec ff7ff ff6b
a ff6b lda ff6f,x
```

und

```
t ff7ec ff7ff 1ff6b
a 1ff6b lda ff6f,x
```

notwendig. Die ersten beiden reichen für einen Versuch auch aus. Eine Anmerkung noch zu den Monitoreingaben: Die führende erste sedezimale Ziffer entspricht in ihrer Bedeutung genau den Konfigurationsindizes. ff7ec bedeutet also die Adresse $f7ec in der Konfiguration, die unter dem Index 15 angegeben ist (Bank 0, alle ROMs eingeschaltet). Dasselbe trifft auf die Banknummer im BASIC-Befehl bank zu. Die angegebene Nummer ist nicht die Nummer der RAM-Bank, sondern der Konfigurationsindex. Insofern ist das Befehlswort vielleicht ein wenig irreführend gewählt worden.

Das Beispiel:

```
c000    lda #$ff     Highbyte von BSOUT
c002    sta 3        nach pchi
c004    lda #$d2     Lowbyte von BSOUT
c006    sta 4        nach pclo
c008    lda #$41     ASCII 'a' als auszugebendes Zeichen
c00a    sta 6        nach areg
c00c    lda $ff00    Sehr wichtig: CR nach JSRCFG !!
c00f    sta $02de
c012    lda #15      BSOUT soll in der Konfiguration mit
c014    sta 2        Index 15 aufgerufen werden (Bank 0, alle ROMs ein)
c016    jsr $02cd    JSRFAR aufrufen
c019    rts          fertig
```

Gestartet wird das Beispiel von BASIC aus durch die Befehle

bank 0: sys dec("c000")

oder vom Monitor aus durch

j c000

Sie sehen, es funktioniert!

2.2 BASIC-Routinen in der Common Area

2.2.1 CHRGET ($380)

Genau wie das Betriebssystem, so kopiert auch das BASIC 7.0 einige wichtige Routinen in den gemeinsamen RAM-Bereich. Diese Routinen sind allerdings nicht in der Hauptsache für den Programmierer gedacht, sondern in ständigem Gebrauch durch den BASIC-Interpreter, was uns nicht hindern soll, sie trotzdem zu benutzen.

Die wichtigste Routine unter ihnen stellt CHRGET (Character Get = Zeichen holen) dar. Jedes einzelne Zeichen des BASIC-Textes wird vom Interpreter mit Hilfe dieser Routine zur Verarbeitung herangeholt.

Vielleicht haben Sie sich schon gefragt, wie es der Interpreter schafft, einen BASIC-Text zu verdauen, der zum größten Teil unter ROMs liegt. Der Text, das BASIC-Programm, darf ja in Bank 0 maximal bis $feff reichen, also noch unter der I/O hindurch bis weit unter das Betriebssystem-ROM. Die Antwort dürfte jetzt klar sein. In ähnlicher Weise wie FETCH holt auch CHRGET ein Zeichen, indem zuerst alle ROMs abgeschaltet werden, das Zeichen geholt wird, und die ROMs wieder eingeschaltet werden.

Gleichzeitig besitzt CHRGET noch einige weitere Eigenschaften, die aus dem Listing ersichtlich werden. Deshalb hier zuerst das Listing von CHRGET:

CHRGET ($380)

```
0380    inc $3d       Textpointer $3d/$3e um 1 erhöhen
0382    bne $0386
0384    inc $3e
0386    sta $ff01     LCRA
0389    ldy #0
038b    lda ($3d),y   Zeichen im Akkumulator laden
038d    sta $ff03     LCRC
0390    cmp #$3a      ist gleich dem ASCII ':'
0392    bcs $039e     größer als ASCII einer Ziffer, dann Ende
0394    cmp #$20      ASCII des Leerzeichens, dann überlesen
0396    beq $0380     nächstes Zeichen betrachten
0398    sec           sonst ASCII von '0'
0399    sbc #$30      subtrahieren
039b    sec           und noch einmal $d0 = $100-$30
039c    sbc #$d0      subtrahieren
039e    rts           fertig
```

Das Listing kann man in dieser Form noch nicht einfach im Raum stehen lassen. Insbesondere die seltsamen Subtraktionen am Ende der Routine müssen noch erklärt werden. Aber fangen wir von vorn an:

CHRGET arbeitet anders als FETCH und die anderen Routinen mit einem festen Pointer

TXTPTR = $3d

Dieser Pointer weist immer genau auf die Stelle innerhalb des BASIC-Programms, die gerade bearbeitet wird. Wer mit dem C 64 vertraut ist, kennt die Entsprechung des Pointers unter der Adresse $7a.
Das erste, was CHRGET bei einem Aufruf unternimmt, ist, diesen Textpointer um 1 zu erhöhen, der Zeiger weist nun auf das nächste Zeichen innerhalb des Textes.
Da es öfter einmal vorkommt, daß ein gerade erst gelesenes Zeichen noch einmal gelesen werden muß, zum Beispiel weil eine Routine von mehreren Stellen aus aufgerufen wird, ohne daß das gelesene Zeichen im Akku übergeben werden kann, erhält der CHRGET-Rest ohne Erhöhung des Textpointers einen eigenen Namen:

CHRGOT = $386

Der nächste Schritt der CHRGET-Routine besteht darin, den Akku in das LCRA zu laden. Wie bei der Beschreibung der MMU nachzulesen, führt ein Schreibzugriff auf das LCRA dazu, daß der Wert des zugehörigen Prä-Konfigurationsregisters PCRA in das Konfigurationsregister übertragen wird. Auch die normale Belegung des PCRA kennen wir: sie ist $3f, alle ROMs sind in Bank 0 ausgeschaltet. Durch den Befehl 'sta LCRA' wird also der gesamte BASIC-Text für CHRGET einsehbar.
Die nächsten beiden Befehle sind dann klar. Es wird das Zeichen aus dem Text geholt. Wer vorher auf dem 64er programmiert hat, wird sich umgewöhnen müssen, da bei der CHRGET-Routine des 128ers das Y-Register benötigt wird. Andererseits können wir uns merken, daß ein CHRGET- oder CHRGOT-Aufruf immer den Wert Null im Y-Register übergibt, man kann also eventuell ein ldy # 0 einsparen. Durch die darauffolgende Operation wird das LCRC beschrieben. Dies führt analog zum Beschreiben des LCRA dazu, daß die vorher ausgeschalteten ROMs wieder eingeblendet werden.
Soweit ist CHRGET nicht sehr unterschiedlich zu FETCH, sieht man davon ab, daß das Verändern des CR hier durch den Einsatz der vorprogrammierten PCRs erfolgt. Jetzt wird es aber interessant, denn das von CHRGET geholte Textzeichen wird untersucht.
Dazu wird es zunächst mit dem Doppelpunkt, dem Trennzeichen zwischen BASIC-Befehlen, verglichen. Ist es gleich dem Doppelpunkt, wird CHRGET mit gesetztem EQUAL-Flag (Zero-Flagge) verlassen.
Ist das Zeichen größer oder gleich dem Doppelpunkt, ist beim Verlassen der Routine außerdem das CARRY-Flag gesetzt. Das gesetzte CARRY bedeutet damit schon einmal,

daß es sich bei dem Zeichen nicht um eine Ziffer handeln kann, denn der Doppelpunkt ist vom ASCII-Wert her genau um 1 größer als die 9.
Die nächsten beiden Befehle sind wieder klar. Leerzeichen innerhalb des Textes werden ganz einfach überlesen.
Dann kommt die etwas seltsame Subtraktion, die nur einen Sinn hat: Es soll erreicht werden, daß das Carry-Flag gelöscht ist, wenn es sich bei dem Zeichen tatsächlich um eine Ziffer handelt. Wir erinnern uns, daß in dieser Abfrage nur noch die Zeichen behandelt werden, die kleiner gleich der Neun sind. Die größeren wurden schon durch den Vergleich mit dem Doppelpunkt abgehandelt. Dieses Ziel ist aber ganz einfach zu erreichen, wenn man sich folgendes überlegt:
Nach der Subtraktion des ASCII-Wertes der Null vom geholten Zeichen stehen wir vor der Situation, daß der Inhalt des Akkumulators größer oder gleich 0 ist, wenn es sich um eine Ziffer gehandelt hat, oder kleiner als Null, wenn es keine Ziffer war. Negative Zahlen kennt der Prozessor aber nicht, dafür ist dann das Carry gelöscht (Unterlauf) und im Akku befindet sich das Zweier-Komplement der negativen Zahl, im vorliegenden Fall eine Zahl zwischen $ff und $d0.
Ziehe ich von dieser Zahl die Zahl $d0 ab, erhalte ich erst einmal die ursprüngliche Zahl wieder, da $d0+$30 gerade gleich $100 ist. Zusätzlich erhalte ich ein Carry, das gesetzt ist, wenn die Ausgangszahl vor der zweiten Subtraktion größer oder gleich $d0 war, ansonsten ist das Carry gelöscht. Eine Zahl größer gleich $d0 erhielten wir nach der ersten Subtraktion aber immer dann, wenn es sich um keine Ziffer handelte. Damit haben wir unser Ziel erreicht. Die CHRGET-Routine liefert uns im Carry-Flag die Information, ob das gelesene Zeichen eine Ziffer ist oder nicht.

Faßt man die Beeinflussung der Flags durch CHRGET einmal zusammen, erhält man folgendes Bild:

CARRY	gelöscht: daraus folgt Ziffer gesetzt: daraus folgt nur, daß es keine Ziffer war
EQUAL	das gelesene Zeichen war entweder ein Nullbyte oder der Doppelpunkt

Ein kleiner, aber sehr beliebter Trick bei der Verwendung der CHRGET-Routine soll nicht unerwähnt bleiben. Er arbeitet mit der Speicherstelle

SKIP = $395

Durch Verändern dieser Speicherstelle kann man erreichen, daß die CHRGET-Routine keine Blanks mehr überliest. Anstelle der Blanks könnte man zum Beispiel den Punkt nach SKIP schreiben, wenn man eine Eingabe aus einer Maske liest, oder man kann erreichen, daß ein INPUT mit führenden Leerzeichen erfolgen kann. Dies alles wohlgemerkt auf Maschinenspracheebene, von BASIC aus ist so etwas nicht zu machen.

2.2.2 Indirekte Laderoutinen des BASIC

Außer CHRGET werden vom BASIC noch fünf weitere kleinere Routinen in die Common Area kopiert. Das Kopieren übernimmt übrigens die Routine COPY2 ($406c). Die kopierten Routinen laden den Akkumulator jeweils mit einem Wert aus Bank 0 oder 1, wobei während des Ladens wie bei CHRGET die ROMs unter Zuhilfenahme der PCRs und LCRs ausgeschaltet werden.

Die ersten beiden dieser Routinen arbeiten mit variablen Pointern. Ihre Listings:

INSUBR0 ($39f)

```
039f    sta $03a6      Akku gleich verwendeter Pointer
03a2    sta $ff01      LCRA: ROMs in Bank 0 ausschalten
03a5    lda ($00),y    Akku laden
03a7    sta $ff03      LCRC: ROMs wieder einschalten
03aa    rts            fertig
```

INSUBR1 ($3ab)

```
03ab    sta $03b2      Akku wieder der Pointer
03ae    sta $ff02      LCRB: Bank 1, alle ROMs aus
03b1    lda ($00),y    Akku laden
03b3    sta $ff04      LCRD: Bank 1, ROMs wieder ein
03b6    rts            fertig
```

Wie man sieht, gleichen sich die beiden Routinen bis auf den Unterschied, daß INSUBR1 sich auf Bank 1 bezieht. Der Interpreter hat es als ROM-Programm erheblich leichter, sich Werte aus anderen Bänken zu beschaffen, da die ROMs einfach in andere RAM-Bänke eingeblendet werden können. Das können wir mit eigenen Maschinenprogrammen natürlich nicht machen. Im Fall von INSUBR1 sieht die Sache so aus: Will der Interpreter einen Variablenwert oder einen String lesen, die beide in Bank 1 liegen, so schaltet er einfach die RAM-Bank unter seiner Sitzfläche um, einmal bildlich gesprochen, und schon hat er Zugang zu Bank 1.

Der Aufruf von INSUBR0 und 1 ist denkbar einfach:

```
lda #pointer        Der Akku enthält das Lowbyte des Pointers
jsr INSUBR          und Aufruf der Laderoutine
```

Die drei letzten Routinen in der Common Area entsprechen INSUBR0 und INSUBR1, nur arbeiten sie mit fest eingestellten Pointern:

I24SR1 ($3b7)

```
03b7    sta $ff02      LCRB: wie INSUBR1
03ba    lda ($24),y    $24/$25 als fester Pointer
03bc    sta $ff04      LCRD
03bf    rts
```

I26SR0 ($3c0)

```
03c0    sta $ff01      LCRA: wie INSUBR0
03c3    lda ($26),y    $26/$27 als Pointer
03c5    sta $ff03      LCRC
03c8    rts
```

INDTXT ($3c9)

```
03c9    sta $ff01      LCRA
03cc    lda ($3d),y    Laden über txtptr
03ce    sta $ff03      LCRC
03d1    rts
```

Besonders die letzte Routine ist interessant, wenn man sich erinnert, daß CHRGET und CHRGOT immer eine Null im Y-Register zurücklassen. Wenn die Untersuchung des Textzeichens auf Ziffer nicht interessiert, kann CHRGOT durch INDTXT ersetzt werden. Dabei werden Leerzeichen natürlich nicht überlesen.

3 Das Betriebssystem des 128ers

Thema: Die Tastatur, Verwaltung des 40- und 80-Zeichen-Bildschirms, Verbindung mit externen Geräten. Die Kernal-Routinen der Sprungleiste $ff47 bis $ffff.

3.1 Die Tastatur

Obwohl meistens nicht als solches angesehen, stellt die Tastatur für den Computer ein Eingabegerät fast wie jedes andere dar, das mit der eigentlichen Funktion des Computers nichts zu tun hat. Wie jedes an den Computer angeschlossene Gerät wird auch die Tastatur erst durch die entsprechende Software einsatzbereit. Um die Wirkungsweise dieser Software, der Kernal-Routinen zur Tastaturabfrage, zu verstehen, müssen wir zunächst einmal nachsehen, wie das Gerät Tastatur aufgebaut ist.

3.1.1 Aufbau und Abfrage der Tastatur

Die Tastatur des 128ers ist in Form einer Matrix von 11 Zeilen und 8 Spalten miteinander verbunden. Jede Taste steht auf einer Kreuzung zwischen einer Matrixzeile und einer Matrixspalte. Drückt man eine beliebige Taste, bewirkt dies, daß eine der Matrixzeilen mit einer der Matrixspalten elektrisch verbunden wird. Wird durch die betreffende Matrixzeile ein Impuls geleitet, so läßt sich dieser in der verbundenen Matrixspalte messen. Dies ist schon das Prinzip der Tastaturabfrage. Es werden ganz einfach alle Matrixzeilen hintereinander mit einem Impuls beschickt und in den Matrixzeilen nachgesehen, ob der Impuls durchgekommen ist oder nicht. Durch die dann bekannte Zeile und Spalte der gedrückten Taste ist diese eindeutig bestimmt.

Die Tastaturmatrix ist hardwaremäßig mit dem CIA 1 verbunden. Die Leitungen der Matrixzeilen 1 bis 8 sind an das Port-Register A des CIA, die acht Matrixspalten an das Port-Register B angeschlossen. Das PRA wird als Ausgang mit den Prüfimpulsen beschickt, das PRB dient als Prüfeingang.
Damit stehen wir auch schon vor dem ersten Problem, das sich den Commodore-Konstrukteuren stellte: Wohin mit den Matrixzeilen 9 bis 11, die im PRA des CIA 1 keinen Platz mehr finden? Man hat sich hier entschlossen, diese Leitungen dem VIC II aufzubürden, der auch sonst schon einige Aufgaben erledigt, die nichts mit seiner eigentlichen Funktion als Videocontroller zu tun haben. Der VIC II hat für diesen Zweck ein gegenüber der 64er-Version neues Register erhalten, die Nummer 47, in dessen Bits 0 bis 2 die restlichen Matrixzeilen-Leitungen untergebracht sind. Das Register 47 stellt damit sozusagen eine Verlängerung des Port-Registers A des CIA 1 dar. Es ist zu erwarten, daß dieses Register es auch im 64er-Modus des C 128 möglich macht, die sonst nicht erkannten Tasten, zum Beispiel des Zehnerblocks, zu benutzen. Mit entsprechender Software natürlich.

Die eigentliche Tastaturabfrage findet alle $^1/_{60}$ Sekunde bei dem durch den Timer A der CIA 1 ausgelösten Interrupt statt. Die Abfrage der Matrixzeilen und -spalten übernimmt die Routine des Kernal-Editors.

KEYSCN = $c55d

Die herausgefundene Nummer der Taste wird in der Zeropage-Stelle

SFDX = $d4

festgehalten. Sie kann zwischen 0 und $58 (88) liegen, der letztere Wert entspricht der Information, daß keine Taste gedrückt war.
Tasten, die erst zusammen mit einer anderen einen ASCII-Wert ergeben, betroffen sind Shift-, Control-, die ALT- und die Commodore-Taste, werden gesondert behandelt. Ihre Tastennummer wird nicht in sfdx abgelegt, sondern in Form eines sogenannten Shift-Musters in der Zeropage-Adresse bei

SHFFLG = $d3

untergebracht. Das Shift-Muster enthält zusätzlich in Bit 4 die Information, ob durch den ASCII/DIN-Schalter die DIN-Tastaturbelegung ausgesucht wurde. Diese Information stammt vom Prozessor-Port (Zeropage-Adresse 1). Ist das Bit 4 von SHFFLG gesetzt, so ist DIN gewählt.

Die Zuordnung zwischen den gedrückten Tasten und dem Wert des Shift-Musters ist:

SHFFLG	gedrückt
% 0000 0000	keine
% 0000 0001	Shift
% 0000 0010	C = (Commodore-Taste)
% 0000 0100	Control
% 0000 1000	ALT (alternative Tastaturbelegung)
% 0001 0000	DIN-Belegung

Kombinationen sind erlaubt.

Die Identifikation der Shift-, Control-, C=-, ALT- und der Stop/Run-Taste geschieht in der Matrixabfrage nicht anhand der Tastennummer, sondern durch Lesen des Tastaturcodes aus der Tastatur-Decodiertabelle 1 (Zeichen ungeshiftet).

Die Tasten werden an ihrem Code erkannt, der wie folgt zugeordnet ist:

Tastenfunktion	Code
ALT	8
Control	4
Run/Stop	3
C=	2
Shift	1

Da ein Vektor in der erweiterten Zeropage auf die Decodiertabelle 1 weist, können wir durch eine Änderung des Vektors und entsprechende Anpassung einer eigenen Tabelle erreichen, daß die angesprochenen Tastenfunktionen auf beliebige andere Tasten verlegt werden. Hierzu müssen die neuen für die obigen Funktionen ausgewählten Tasten mit den oben beschriebenen Tastencodes unterlegt werden. Soll zum Beispiel die Shift-Funktion auf die Plustaste des Zehnerfeldes der Tastatur umgebogen werden, brauchen wir hierzu lediglich eine Tabelle, in der der Tastennummer 73 (von Null an gezählt = »+« der Zehnertastatur) der Tastencode 1 zugeordnet ist, eine Tabelle also, deren 73. Eintrag gleich 1 ist.

Die den einzelnen Tasten zugeordneten Tastennummern und Tastencodes der Tabelle 1 finden Sie auch im Anhang.

Der zuständige Vektor in der erweiterten Zeropage liegt bei

KEYTAB = $33e

Er weist normalerweise auf die ROM-Adresse $fa80.

3.1.2 Die Auswertung der Tastaturabfrage

Nach dem inneren Kern der Tastaturabfrage in KEYSCN muß das Ergebnis der Abfrage natürlich noch ausgewertet werden. Bekanntlich erwartet der Benutzer, daß auf bestimmte Tastendrücke einige Funktionen sofort ausgeführt werden. Dazu gehört zum Beispiel die Umschaltung des Zeichensatzes durch gleichzeitiges Drücken von Shift und C=, oder das Betätigen der Taste NO-SCROLL.

Dazu wird nach KEYSCN in die Routine

KEYLOG = $c5e1

übergegangen, die die weitere Bearbeitung übernimmt.

Als erstes wird die NO-SCROLL-Taste abgefragt (Tastencode $57). Je nach Zustand des Flags

LOCKS = $f7

wird eine Pause eingeleitet oder nicht. Bit 6 des Flags ist für eine Sperrung der Pause zu setzen, die NO-SCROLL-Taste hat dann keine Wirkung. Da die NO-SCROLL-Taste sowohl für das Einleiten der Pause als auch für deren Aufhebung zuständig ist, wird das Flag

NSCFLG = $a21

bei Drücken der Taste jeweils invertiert. Ein Wert ungleich 0 entspricht dem Setzen einer Pause. Das Setzen der Pause kann außer durch die NO-SCROLL-Taste auch durch die Tastenkombination Control plus »s« geschehen. Die Pause kann mit dieser Tastenkombination aber nicht wieder aufgehoben werden.

Das Bit 7 des LOCKS-Flags ist zuständig für ein Sperren der Zeichensatzumschaltung mittels Shift + Commodore. Ist es gesetzt, kann keine Umschaltung erfolgen.

Um ein sehr schnelles Umschalten der Zeichensätze und eine schnelle Hintereinanderfolge von Pauseimpulsen zu verhindern, wird die Speicherstelle

SDELAY = $a25

als Zähler benutzt. Eine Aktion erfolgt erst dann, wenn SDELAY gleich Null ist. Nach jeder erfolgten Umschaltung wird SDELAY mit 8 initialisiert. Weitere Durchläufe der

Interruptroutine teilen diesen Wert immer durch 2, so daß sich im Endeffekt eine Verzögerung ergibt, die der Geschwindigkeit der Tastenwiederholung (siehe unten) entspricht.
Liegt weder die Pause-Funktion noch eine Umschaltung auf einen anderen Zeichensatz vor, wird anhand des Shift-Musters SHFFLG eine Tastatur-Decodiertabelle aus 6 möglichen ausgewählt. Die Auswahl der Tabelle geschieht auch hier wieder über Vektoren, die in der erweiterten Zeropage liegen. Den einzelnen Shift-Mustern, das heißt den einzelnen angewählten Tastaturbelegungen, entspricht die folgende Tabellenzuordnung:

Zeichen mit	Vektor auf Decodiertabelle
DIN-Tastatur	$348/$349 ($fbe4)
ALT	$346/$347 ($fa80)
Control	$344/$345 ($fb8b)
C=	$342/$343 ($fb23)
Shift	$340/$341 ($fad9)
normal	$33e/$33f ($fa80)

Die Werte in Klammern geben die Normalinhalte der Vektoren an.

Wie man sieht, kann durch Veränderung dieser Vektoren leicht eine völlig freie Gestaltung der Tastaturbelegung erfolgen.

Aus der festgestellten Decodiertabelle wird entsprechend der Tastennummer der zugehörige ASCII-Wert ermittelt, der der gedrückten Taste entspricht. Ein gelesener Wert von $ff steht für eine nicht zulässige Tastenkombination. Beim Auftreten einer solchen Kombination wird keine Fehlermeldung angesprungen oder sonst etwas. Unzulässige Zeichen werden einfach ignoriert.

3.1.3 Die Tastenwiederholung (REPEAT)

Wie bekannt, werden alle Tasten des C128 im 128er-Modus wiederholt, wenn sie gedrückt gehalten werden. Dies wird erreicht, indem die momentan ermittelte Tastennummer mit der zuletzt ermittelten verglichen wird. Die Speicherung der zuletzt ermittelten Tastennummer erfolgt in der Speicherstelle

LSTX = $d5

Sind beide Nummern gleich, so wird dieses zum Anlaß genommen, zu prüfen, ob eine Tastenwiederholung erlaubt ist oder nicht. Das dafür zuständige Flag ist

RPTFLG = $a22

RPTFLG kann drei verschiedene Zustände annehmen:

RPTFLG	wiederholte Tasten
$80	alle Tasten
$40	keine Taste
0	nur die Tasten DEL, INST, die Cursortasten und die Leertaste

Die Tastenwiederholung zerfällt in zwei Phasen. Die erste Phase ist ein Warten, bis der Repeat-Verzögerungszähler

DELAY = $a24

auf Null heruntergezählt ist. Der Normalinhalt dieses Zählers beträgt $10 (16). Die Routine KEYLOG muß also 16mal mit der gleichen gedrückten Taste angesprungen werden, bis eine Tastenwiederholung einsetzt. Dies macht durchaus einen Sinn, wenn man bedenkt, wie schnell der Prozessor arbeitet. Kein Mensch kann so schnell eine Taste wieder loslassen, wie der Prozessor schon wieder in der nächsten Interrupt-Tastaturabfrage gelandet ist.

Nachdem DELAY auf Null gezählt ist, setzt die Tastenwiederholung ein. Sie erfolgt mit der Geschwindigkeit, die in

KOUNT = $a23

festgelegt ist. Der Normalwert für dieses Repeat-Geschwindigkeits-Flag ist 4, bei jedem vierten Anlaufen von KEYLOG wird also der Code der gedrückten Taste dann auch weitergegeben.

Eine Änderung von DELAY oder KOUNT hat keine Wirkung, da diese Werte innerhalb der Interruptroutine ständig neu initialisiert werden.

3.1.4 Der Tastaturpuffer ($34a)

Schließlich muß KEYLOG noch das erkannte Zeichen in irgendeiner Form aufbewahren, so daß es von anderen Routinen weiterverarbeitet werden kann. KEYLOG wird ja in einem Interrupt aufgerufen und hat keine Verbindung mit anderen Routinen außerhalb der Tastaturabfrage.

Die Sicherung der von KEYLOG erkannten Tastatureingaben erfolgt in einem eigenen Tastaturpuffer

KEYBUF = $34a

Die maximale Länge des Tastaturpuffers wird in

XMAX = $a20

vorgegeben. Normalerweise ist diese Pufferlänge auf 10 voreingestellt. Die ASCII-Codes der durch KEYLOG von der Tastatur geholten Zeichen werden in Form einer Warteschlange in den Puffer geschrieben, das zuletzt geholte Zeichen kommt immer an das Ende der Warteschlange. Ist der Puffer voll, werden keine Zeichen mehr angenommen. Aus der Warteschlange wird ein Zeichen gelesen, indem das erste Zeichen der Schlange ausgelesen und der Rest des Puffers nach vorn aufgerückt wird.
Die Länge der Warteschlange wird im Zeiger

NDX = $d0

aufbewahrt. NDX zeigt immer auf die nächste freie Position im Tastaturpuffer. Ist NDX gleich Null, ist der Puffer leer, ist NDX gleich XMAX, dann ist der Puffer randvoll.

3.1.5 Behandlung der Funktionstasten

Die Funktionstasten werden einer Sonderbehandlung unterzogen, da das Betriebssystem erlaubt, diesen Tasten einen Tasten-String zuzuordnen. Handelt es sich bei einer erkannten Taste um eine der Funktionstasten

F1 - F8
HELP (F9)
Shift RUN/STOP (F10),

so erscheint der Code der Funktionstaste nicht im Tastaturpuffer. Statt dessen wird die Länge des der Taste zugeordneten Strings in die Speicherstelle

KYNDX = $d1

geschrieben. Die Adresse des Funktionstasten-Strings wird in

KEYIDX = $d2

festgehalten. KEYIDX ist ein Zeiger relativ zur Anfangsadresse des Funktions-String-Speichers

KEYSTR = $100a

In diesem Speicher sind die Strings der einzelnen Funktionstasten hintereinander aufbewahrt. Im vorhergehenden Speicher

KEYLEN = $1000

sind die Längen der Strings gespeichert. Die Länge des zu F1 gehörenden Strings in $1000, die Länge des zweiten Strings in $1001 usw.
Aus der Tatsache, daß der Zeiger in den Speicher KEYSTR nur aus einem Byte besteht, erkennt man, daß die Gesamtlänge aller Funktionstasten-Strings nicht größer als 255 sein kann. Da hinter KEYSTR bei $1100 noch ein wichtiger Speicherbereich anzutreffen ist, sollte die Länge sogar 245 nicht überschreiten.
Sollen Zeichen aus dem Tastaturpuffer geholt werden, zum Beispiel durch die Routine GETIN ($ffe4), so haben die Zeichen von Funktionstasten-Strings Vorrang gegenüber den Zeichen im Tastaturpuffer. Mit jedem aus einem String geholten Zeichen wächst der Zeiger KEYIDX und nimmt der Zeiger KYNDX ab, bis er Null ist. Dies entspricht der Tatsache, daß alle String-Zeichen ausgegeben wurden.

Programmierung der Funktionstasten (PFKEY, $ff65)

Die Programmierung der Funktionstasten, die Zuweisung eines Tasten-Strings an eine Funktionstaste, ist denkbar simpel gelöst worden.

Man benötigt hierzu lediglich drei Byte in der Zeropage und die Routine

PFKEY = $ff65

Diese Routine überträgt einen eigenen String in die Bereiche KEYLEN und KEYSTR. Der eigene String darf dabei in einer beliebigen Speicherkonfiguration stehen. Der Konfigurationsindex wird im dritten Byte in der Zeropage an PFKEY übergeben, die Adresse des Strings in den ersten beiden Bytes in der Reihenfolge Low-/Highbyte. Die Länge des neuen Tasten-Strings wird beim Aufruf von PFKEY im Y-Register erwartet, die Nummer der Funktionstaste in X. Der Akkumulator enthält den Zeiger auf die frei gewählten drei Byte in der Zeropage, die die weiteren Angaben enthalten. Ein Beispiel für eine Tasten-String-Zuweisung finden Sie in Kapitel 3.5.

Die bei der Tasten-String-Übertragung verwendeten Zeropage-Adressen bieten die Gelegenheit, den Editor-Hilfsspeicher-Bereich vorzustellen. Alle Speicherstellen dieses Bereiches sind rein für Zwischenspeicherungen vorgesehen, die während des Betriebs des Kernal-Editors anfallen. Die Bytes sind deshalb sehr gut dafür geeignet, in eigenen Maschinenprogrammen ebenfalls als Hilfsspeicher zu dienen. Der genannte Bereich liegt von $da (SEDSAL) bis $df (SEDT2).

3.1.6 DIN/ASCII-Umschaltung, Akzentzeichen

Mittels des ASCII/DIN-Schalters kann die Tastenbelegung jederzeit zwischen einer amerikanischen Belegung und einer DIN-Belegung umgeschaltet werden.
Die Betätigung des Schalters wird innerhalb der Routine KEYSCN anhand des Zustandes des Datenregisters des Prozessorports erkannt und in Bit 4 des Shift-Musters festgehalten. Die Umschaltung selbst wird dann beim nächsten Aufruf der Interruptroutine ausgeführt, wenn sie nicht durch das Bit 7 des Flags

DINFLG = $ac5

verhindert wird. Die Umschaltung selbst besteht darin, daß ein Satz von 6 Vektoren in die erweiterte Zeropage kopiert wird. Den beiden möglichen Belegungen sind die Vektoren zugeordnet, die im ROM unter folgenden Adressen zu finden sind:

ASCII	$c06f
DIN	$fe34

Ist die DIN-Belegung ausgesucht, so besteht die Möglichkeit, Akzentzeichen einzugeben. Die Behandlung der Akzentzeichen in der Kernal-Routine $fcc6 (ist eine Weiterführung von KEYLOG) verläuft folgendermaßen:
Es wird geprüft, ob sich in den Decodiervektoren die Zeiger auf die DIN-Tabellen befinden. Ist dies der Fall und das DINFLG nicht gesetzt, wird das von der Tastaturmatrix geholte Zeichen darauf geprüft, ob es sich um einen Akzent handelt. Trifft dies zu, wird der Code des Akzentes in DINFLG festgehalten. Dies bewirkt, daß die nächsten von der Tastatur geholten Zeichen auf ihre Zulässigkeit in Verbindung mit dem zuerst gefundenen Akzent überprüft werden. Nicht zugelassen sind zum Beispiel Steuerzeichen. Dies geht so lange, bis ein im Rahmen der ROM-Tabellen – die dafür zuständig sind – zulässiges Zeichen eingegeben wurde. Anhand einer zweiten Tabelle wird dann überprüft, ob die DIN-Decodiertabelle einen ASCII-Wert enthält, der einem akzentuierten Zeichen der gefundenen Zusammensetzung entspricht. Je nachdem, ob dies der Fall ist oder nicht, wird als Ergebnis der Akzentzusammensetzung entweder das akzentuierte Zeichen oder das zuletzt geholte Zeichen ohne Akzent ausgegeben.

3.1.7 Einige Anmerkungen zu den Tastaturroutinen

Es ist bekanntlich nicht ganz einfach, lange Maschinenprogramme vollkommen fehlerfrei zu schreiben. Als hätten die Autoren des Kernals dies gewußt, sind in den Routinen zur Tastaturauswertung und auch zur Bildschirmausgabe einige Ungereimtheiten enthalten, die sich einer logischen Erklärung geschickt entziehen.
Als erstes wäre hier das Vorhandensein der sechsten Decodiertabelle zu nennen, der Tabelle, die bei gesetztem Bit 4 des Shift-Flags zur Decodierung herangezogen wird. Wird die ASCII/DIN-Taste geschaltet, so wird zur Zeichendecodierung auf diese Tabelle umgeschaltet. Die eigentliche ASCII/DIN-Umschaltung erfolgt aber schon im nächsten Interrupt, so daß diese Tabelle nur für gleichzeitig mit der Umschaltung eingegebene Zeichen von Bedeutung ist. Für diesen Zweck hätte es die Normaltabelle auch getan, wie es bei der Wahl der DIN-Tastatur auch gemacht wird. Hätte man dies in der Software berücksichtigt, hätte man einen Vektor und eine Tabelle im ROM gespart. Schaut man sich die besagte Tabelle einmal genauer an, zeigt sich auch, daß es sich um eine Unsinnstabelle handelt. Legen Sie hierzu einmal den Tabellenzeiger $fbe4 in den Vektor $346/$347 für Zeichen, die zusammen mit der ALT-Taste eingegeben werden. Sie können dann selbst sehen, wie die Tasten durch diese Tabelle unterlegt werden.
Das zweite, was sofort auffällt, ist, daß die Pausenschaltung mit Control + s mit der Ausgabe der Controlcodes kollidiert. Control + »s« ist in den Ausgaberoutinen für Zeichen mit Control dazu vorgesehen, den gleichen Code wie HOME zu erzeugen. Dies geht natürlich nicht mehr.
Beides sind natürlich keine großen Sachen, aber zusammen mit der Tatsache, daß die KEYLOG-Routine über das ROM verstreut wurde, und daß mehrmals vollkommen unsinnige Befehle in der KEYLOG-Routine auftauchen, die allerdings die Funktionstüchtigkeit nicht beeinflussen, bietet die Routine kein besonders schönes Bild.

3.2 Die Bildschirmverwaltung des Betriebssystems

Die Kernal-Routinen sind in der Lage, sowohl den VIC- als auch den VDC-Textschirm zu bedienen. Eine Fensterverwaltung innerhalb der Textschirme gehört ebenso zum Repertoire der Bildschirmverwaltung wie die Aufteilung des VIC-Bildschirms in ein Grafik- und ein Textfenster. Diese sogenannte Splitscreen-Option ist also keine BASIC-Funktion, sondern schon Bestandteil des Betriebssystems!
Insgesamt ist die Bildschirmverwaltung mit all ihren Optionen sicher der komplexeste Teil des Betriebssystems, aber gerade durch die Komplexität ist sie auch einer der interessantesten Teile, der dem Programmierer viel Spielraum für seine Phantasie in eigenen Programmen läßt.

Um den regelrechten gordischen Knoten zu lösen, den die Routinen des Kernal-Editors bilden, sehen wir uns als erstes am besten an, was im Untergrund des Kernals, in den Interrupts, vor sich geht und mit den Bildschirmen zusammenhängt.

3.2.1 Splitscreens

In der gleichen Interruptroutine, die schon zur Tastaturmatrix-Abfrage verzweigte, behandelt das Kernal auch die einstellbare Aufteilung des VIC-II-Bildschirms in ein Grafik- und ein Textfenster. Das Textfenster liegt dabei immer unterhalb des Grafikbereiches, dies ist vielleicht von den entsprechenden BASIC-Befehlen her bekannt.

Die Einstellung der Bildschirmaufteilung kann in der Adresse

GRAPHM = $d8

vorgenommen werden. Die Zuordnung der verschiedenen Modi ist wie folgt:

GRAPHM	gewählter Bildschirm
% 0000 0000	gesamter Bildschirmtext
% 0010 0000	gesamter Schirm Einzelpunkt-Grafik
% 0110 0000	Splitscreen Einzelpunkt/Text
% 1010 0000	gesamter Schirm Multicolor
% 1110 0000	Splitscreen Multicolor/Text

Ist die Option des Splitscreen gewählt, muß in der Speicherstelle

SPLIT = $a34

die Rasterzeile angegeben werden, innerhalb derer die Grenze zwischen Grafik- und Textteil verlaufen soll. Die Adresse ist übrigens im Handbuch falsch angegeben.
Ein Splitten des VDC-Bildschirms in der gleichen Art ist natürlich nicht möglich, da der VDC von der Hardware her nicht die Möglichkeit eines Rasterzeilen-IRQ besitzt.
Die Wirkung der beiden beschriebenen Speicherstellen können Sie auch sehr schön mit Hilfe des bordeigenen Monitors verfolgen. Spielen Sie ruhig einmal mit ihnen herum!
Die Umschaltung des VIC II zwischen zwei Modi bedingt selbstverständlich, daß Registerinhalte des VIC verändert werden. Für die eigentliche Modus-Umschaltung war schon GRAPHM zuständig. Je nach Einstellung von GRAPHM werden die Steuerregister des VIC hin- und hergeschaltet. Dies allein genügt aber nicht, wenn man bedenkt, daß zum Beispiel das Video-RAM des VIC im Grafikmodus als Farb-RAM dient. Aus

diesem Grund wird auch das VIC-Register 24 (Startadressen, Video-RAM und Character-ROM beziehungsweise Grafik) bei den Rasterzeilen-IRQs umgeschaltet. Die beiden Zustände des Registers werden in

VM1 = $a2c für das Textfenster

und

VM2 = $a2d für das Grafikfenster

festgehalten und können dort verändert werden (siehe Register 25 des VIC II).

Ein weiteres Flag ist notwendig, damit das Betriebssystem erkennen kann, ob der VIC mit einem eigenen Zeichensatz oder mit dem ROM-Zeichensatz betrieben werden soll. Dieses Flag befindet sich in der Zeropage-Adresse

CHAREN = $d9

Es ist nur das Bit 1 umschaltbar, die anderen Bits müssen unbedingt immer auf Null gehalten werden! Ist das Bit 1 gesetzt, befindet sich der Zeichensatz im RAM.
Durch den Aufbau der Interruptroutine bedingt werden Änderungen oder Einstellungen in GRAPHM oder VM1 beziehungsweise VM2 und auch in SPLIT wirksam, ohne daß die VIC-Register explizit beschrieben werden müssen. Das ist natürlich eine sehr schöne Sache.

3.2.2 Die Textschirme, Windows

Die Ein- und Ausgabe vom oder auf den Textschirm entspricht ganz einfach dem Lesen oder Beschreiben des gerade zuständigen Video-RAMs sowie des Farb- oder Attribut-RAMs.
Um die RAMs beschreiben und lesen zu können, müssen die Basisadressen der Speicher für das Kernal verfügbar sein. Es kann schließlich nicht bei jeder Ausgabe in den Registern der Videocontroller nachsehen, ob zum Beispiel ein Video-RAM zwischenzeitlich verschoben wurde.
Aus diesem Grund werden die Startadressen der benötigten RAMs in den unten aufgeführten Speicherzellen bereitgehalten:

VM3 = $a2e Highbyte der Startadresse des VDC-Video-RAMs
VM4 = $a2f Highbyte VDC-Attribut-RAM
VM5 = $a3b Highbyte des VIC-Video-RAM-Starts

Der gerade eingestellte Textschirm, ob VIC oder VDC, wird im Flag

MODE = $d7

vermerkt. Das Bit 7 von Mode ist gesetzt, wenn der VDC-Schirm angeschaltet ist.
Der in das Betriebssystem integrierte Kernal-Editor, zuständig für die Verwaltung der Bildschirme, faßt jeden Textschirm, auch den gesamten, als Textschirmfenster auf. Ein Textschirmfenster entspricht einem festgelegten Ausschnitt des jeweiligen Video-RAMs (+ Farb-RAM/Attribut-RAM). Die Grenzen eines solchen Windows werden in den Speicherstellen

Name	Adresse	
SCBOT	**$e4**	Untergrenze
SCTOP	**$e5**	Obergrenze
SCLF	**$e6**	linke Grenze
SCRT	**$e7**	rechte Grenze

festgehalten. Der Einfachheit halber sind die Grenzen eines Fensters nicht in absoluten Video-RAM-Adressen, sondern in Zeilen und Spalten angegeben – damit sind die Speichergrenzen sowohl für das Video-RAM als auch für das Attribut-RAM festgelegt.
Die aktuelle Cursor-Position wird, wieder angegeben in Zeilen und Spalten, in

TBLX	**$eb**	Zeile
PNTR	**$ec**	Spalte

aufbewahrt.

Die Verbindung zwischen der in Zeilen und Spalten angegebenen Position innerhalb des Video-RAMs und der wirklichen Adresse, die man benötigt, um dann ein Zeichen in das RAM schreiben oder aus dem RAM lesen zu können, geschieht über die Adressen

PNT = $e0 (/$e1)	Zeiger in das aktuelle Video-RAM
USER = $e2 (/$e3)	Zeiger in das Farb- oder Attribut-RAM

Diese beiden Zeiger enthalten immer die Startadresse der aktuellen Zeile im Video-RAM beziehungsweise im Farb- oder Attribut-RAM, also die Position der Spalte 0 der Textschirmzeile im RAM.
Der Gesamtbildschirm wird als Fenster mit der Anzahl von Zeilen und Spalten angesehen, die in den beiden Speicherstellen

LINES = $ed — Maximale Anzahl Zeilen

und

COLUMNS = $ee — Maximale Anzahl Spalten

festgelegt sind. Die Normalwerte für diese Bytes sind (24,39) im 40er-Bildschirm und (24,79) im 80er-Modus. Durch Veränderung dieser Werte kann man ein nicht mehr aufhebbares Fenster schaffen. Jedenfalls ist es nicht durch zweimaliges HOME aufzuheben wie die anderen Fenster.

3.2.3 Umschaltung zwischen den Textschirmen

Ein besonderer Leckerbissen des Betriebssystems ist, daß jederzeit zwischen dem 40- und dem 80-Zeichen-Textschirm umgeschaltet werden kann, ohne daß bei der Umschaltung der Inhalt des vorherigen Bildschirms verlorengeht. Sogar eingestellte Fenster, die Cursor-Position und gesetzte Tabulatorstopps bleiben bei der Umschaltung erhalten. Schaltet man auf den ersten Bildschirm zurück, erhält man ihn im Originalzustand wieder.
Dies wird erreicht, indem bei jedem Umschalten der sogenannte aktive Bildparameter-Speicher mit dem passiven Parameter-Speicher ausgetauscht wird. Die beiden Speicherbereiche unterteilen sich in zwei Teilbereiche:

ACTPAR **$e0 - $fa**
PASPAR **$a40 - $a5a**

ACTMSK **$354 - $360**
PASMSK **$a60 - $a7d**

In ACTPAR werden die Parameter des aktuell ausgesuchten Bildschirms untergebracht (Zeichenfarbe, Cursor-Position, Fenstergrenzen ...), im passiven Speicher befinden sich in derselben Anordnung wie im aktiven Speicher die Parameter des anderen Bildschirms.
ACTMSK enthält in den ersten 10 Byte die gesetzten Tabulatorstopps in Form einer Bitmap. Jedes gesetzte Bit dieser 10 Byte entspricht einem gesetzten Stopp. Das Bit 7 des ersten Byte entspricht dabei der Spalte 0, das Bit 7 des zweiten Byte dann der Spalte 8 usw. Es können maximal 8 solcher Stopps gesetzt werden.
Die restlichen drei Byte des Bereiches enthalten die sogenannten Zeilenüberläufe. Sie sind in derselben Weise codiert wie die Tabulatorstopps. Die Erklärung der Zeilenüberläufe ist recht einfach: Wie bekannt, darf eine BASIC-Eingabezeile 160 Zeichen lang sein. Dies ist länger als eine Bildschirmzeile lang ist, folglich muß man bei der Eingabe einer kompletten Eingabezeile mehrere Bildschirmzeilen verwenden. Ein gesetztes Bit in der Zeilenüberlauf-Bitmap bedeutet nun nichts weiter, als daß die korrespondierende Bildschirmzeile eine Fortsetzungszeile einer Eingabezeile ist. Mit anderen Worten, in diese Bildschirmzeile ist man gelangt, indem man über den rechten Rand der vorhergehenden Zeile hinweggeschrieben hat. Das Überlauf-Bit ist wichtig, wenn die Eingabezeile mit einem Return abgeschlossen wird. Nur anhand des Überlauf-Bits kann erkannt werden, wo der Start der Eingabezeile tatsächlich zu suchen ist.

Zeile 0 entspricht in der Zeilenüberlauf-Bitmap dem Bit 7 des ersten Byte der Map, die zweite Zeile dem Bit 6 usw.
Arbeitet man mit Bildschirm-Fenstern, ist es wichtig, darauf zu achten, daß bei einer Neudefinition der Fenstergrenzen die Zeilenüberlauf-Bits gelöscht werden, sollen innerhalb des Fensters Eingaben gemacht werden. Dies geschieht normalerweise schon dadurch, daß eine neue Fenstergrenze mit Hilfe der Kernal-Routine

WINDOW = $c02d

definiert wird. Es reicht also nicht einfach aus, eine neue Fenstergrenze in die entsprechende Speicherzelle zu schreiben.

3.2.4 Die Zeichenausgabe

Nachdem die Grundzüge des Bildschirmaufbaus nun bekannt sind, sehen wir uns einmal an, wie der Kernal-Editor bei der Zeichenausgabe auf den Bildschirm im einzelnen vorgeht.

Das auszugebende Zeichen wird der Kernal-Routine

PRINT = $c00c

im Akku übergeben. Die PRINT-Routine ist die Unterroutine des allgemeineren BSOUT, die die Bildschirmausgabe übernimmt.
PRINT legt das auszugebende Zeichen als erstes zur Sicherung in

DATAX = $ef

ab. Dort steht es den vielen Unterroutinen von PRINT zur Verfügung. Je nach Zustand des Pausen-Flags NSCFLG, das während der IRQ-Tastaturabfrage gesetzt werden kann, hält PRINT jetzt an und wartet, bis das Flag gelöscht ist. Das Flag CRSW ($d6), über das bei der Eingabe vom Bildschirm noch zu sprechen sein wird, wird durch die Bildschirmausgabe nun gelöscht.

Betrachten wir jetzt zuerst den Fall, daß das auszugebende Zeichen gleich RETURN ($0d, 13) ist. Bekanntlich hat RETURN eine Sonderrolle inne, es schließt eine eventuelle Eingabe ab.

Durch die Ausgabe von RETURN werden einige Flags gelöscht, die in nachstehender Tabelle aufgeführt sind:

Name	Adresse	Funktion
COLOR	**$f1**	Attribut (Schriftfarbe) des nächsten auszugebenden Zeichens. Durch RETURN werden die Bits 5 und 4 gelöscht, das entspricht beim 80-Zeichen-Modus den Attributoptionen UNDL und FLASH (s. Beschreibung des Attribut-RAMs). Im 40-Zeichen-Modus haben diese Bits keine Bedeutung, denn die Schriftfarbe kann nur einen Wert von 0 bis 15 im Farb-RAM annehmen.
RVS	**$f3**	Das REVERSE-Flag. Ein Wert ungleich Null in RVS zeigt an, daß das nächste Zeichen revers ausgegeben werden soll. RVS wird beispielsweise durch die Kombination Control + 9 (Rev On) gesetzt.
QTSW	**$f4**	Das QUOTE-Flag, zu deutsch das Anführungszeichen-Flag. Dieses Flag bewirkt, daß Steuerzeichen wie zum Beispiel Cursor-Up oder Cursor-Down als reverse Zeichen in Strings eingeschlossen werden können. Ein Wert ungleich Null entspricht einem gesetzten Quote-Switch.
INSRT	**$f5**	Der Zähler für eingegebene Inserts. Ist der Zähler noch ungleich Null, so sind noch Zeichen einzufügen. Dies wirkt sich ähnlich wie bei QTSW in einer Unterdrückung der Ausführung von Steuerzeichen aus.

Jetzt kennen wir schon den Großteil der Flags, die die Bildschirmausgabe beeinflussen. Wie und wodurch die einzelnen Flags und Schalter gesetzt werden, möchte ich als bekannt voraussetzen, da man ja ständig mit dem Kernal-Editor arbeitet.
Das Löschen der obigen Flags wird nicht nur durch RETURN ausgelöst. Auch die Ausgabe von SHIFT-RETURN und die Kombination ESC + »o« führt zur Löschung.

Da wir gerade bei Flags sind, können wir auch die restlichen drei Flags gleich noch abhandeln. Es sind dies:

INSFLG $f6
SCROLL $f8

und

BEEPER $f9

BEEPER verhindert die Ausgabe eines Signaltons durch die Kombination Control + »g« (ASCII-Wert 7). Das gesetzte Bit 7 des Flags sorgt dafür, daß nervenschonende Ruhe eintritt.

SCROLL bewirkt, daß ein Aufwärts-Scrollen des Bildschirms verhindert wird, wenn das höchstwertige Bit des Flags gesetzt ist. Erreicht der Cursor den unteren Bildschirmrand, wird nicht der Bildschirm nach oben gescrollt, sondern der Cursor auf den oberen Bildschirmrand gesetzt. Dies kann eventuell nützlich werden, wenn mit Eingabemasken gearbeitet wird. Das Scrollen durch andere Ursachen, etwa durch die Eingabe von ESC + »v«, wird durch SCROLL nicht verhindert. Gesetztes Bit 6 des Flags verhindert, daß die letzte Bildschirmzeile versehentlich ein Zeilenüberlauf-Bit erhält. Normalerweise wird das durch SCROLL verhinderte Scrollen ja auch durch ein Voll- und Überschreiben der letzten Zeile hervorgerufen, so daß die hereingescrollte Bildschirmzeile eine Fortsetzungszeile ist, was im zugehörigen Zeilenüberlauf-Bit vermerkt wird.

INSFLG ist für den automatischen Einfügemodus zuständig, der durch ESC + »a« eingeschaltet und durch ESC + c wieder ausgeschaltet werden kann. Eingeschaltet enthält INSFLG den Wert $ff (255), ein gesetztes Bit 7, also $80 (128), reicht aber auch, ausgeschaltet ist es 0.

Damit wären auch schon alle Flags, die die Bildschirmausgabe beeinflussen, behandelt. Es sind eine ganze Menge, aber davon sollte man sich nicht abschrecken lassen. Normalerweise hat man mit den Interna des Kernal-Editors natürlich nichts zu tun. Statt zum Beispiel das REV-Flag in der richtigen Weise zu setzen, wird man einfach den Code für Revers-On über PRINT ausgeben. Alles weitere ist dann die Sache der Kernal-Routinen. Zum besseren Verständnis der entsprechenden Befehle, und damit auch zur richtigen Anwendung derselben, ist es aber auf jeden Fall besser, man weiß, was im Editor im einzelnen abläuft. Und sei es aus dem Grund, daß man vermeiden kann, die für die Routinen wichtigen Speicherstellen zu überschreiben.

Zurück zur eigentlichen Zeichenausgabe. Anhand der Ausgabe des RETURN haben wir die Flags kennengelernt. Jetzt bleibt noch zu klären, was mit den übrigen Zeichen geschieht.

Hier lassen sich ein paar Fälle unterscheiden:

1. Es kann sich um die Ausgabe von Control-Codes handeln.
2. Es sind Escape-Sequenzen.
3. Es sind normale Steuerzeichen wie HOME, DEL usw.
4. Es sind gewöhnliche Zeichen im ASCII-Bereich unter 128 (ungeshiftet).
5. Es sind geshiftete Zeichen.

Warum diese Unterscheidung so ausdrücklich an dieser Stelle getroffen wird, wird bei der Betrachtung der Kernal-Vektoren in der erweiterten Zeropage noch klar werden. Die Ausgabe der unter den Punkten 1, 2 und 5 aufgeführten Zeichen läßt sich nämlich noch beeinflussen!

Auf die einzelnen Routinen, die der Ausgabe der verschiedenen Zeichen dienen, möchte ich nicht weiter eingehen. Dies würde den Rahmen des Buches ganz einfach sprengen. Es sei nur noch die letzte für die Zeichenausgabe wichtige Speicherstelle angesprochen:

LSTCHR = $f0

Hier wird das letzte ausgegebene Zeichen aufbewahrt. Anhand dieses Bytes kann auch erkannt werden, ob eine Escape-Sequenz ausgegeben werden muß.

3.2.5 Eingaben vom Bildschirm

Bei der Zeicheneingabe vom Bildschirm, also bei der Benutzung des Bildschirms als Eingabegerät, sind zwei Fälle zu unterscheiden:

A) Das Lesen des Video-RAMs, wobei der Cursor ausgeschaltet ist. Das entspricht in BASIC einer Befehlsfolge wie

```
open 1,3: input#1, a$
```

B) Die Eingabe, wobei der Cursor eingeschaltet ist. Hier werden die Zeichen erst von der Tastatur geholt und auf den Bildschirm geschrieben, und zwar so lange, bis ein Carriage Return eingegeben wurde. Erst dann werden Zeichen aus dem Video-RAM geholt. Dieser Vorgang ist häufiger, er entspricht dem normalen

```
input a$
```

Betrachten wir zunächst den ersten Fall.

3.2.5.1 Lesen vom Bildschirm bei ausgeschaltetem Cursor

Dieser Lesevorgang ist dadurch gekennzeichnet, daß das Flag

CRSW = $d6

ungleich Null, aber das Bit 7 des Flags nicht gesetzt ist. Normalerweise steht eine 3, die Gerätenummer des Bildschirms, in CRSW, wenn auf die betreffende Art gelesen werden soll.

Das Lesen selbst ist dann außerordentlich einfach:
Aus dem Video-RAM wird das Zeichen geholt, das sich an der aktuellen Cursor-Position befindet, wobei der Cursor natürlich normalerweise nicht sichtbar ist. Aus dem Farb-RAM oder Attribut-RAM wird ebenfalls das entsprechende Byte geholt. Das gelesene Zeichen wird in DATAX ($ef) abgelegt, die Farbe beziehungsweise das Attribut wandert in den Zwischenspeicher

TCOLOR = $f2

Anschließend wird der Cursor um eine Position nach rechts bewegt, wenn es noch möglich ist. Die Eingabe wird nämlich von zwei Grenzen aufgehalten:

INDX	**= $ea**	letzte Spalte für das Lesen
LINTMP	**= $a30**	letzte Zeile für die Eingabe

Nachdem die aktuelle Cursor-Zeile gleich LINTMP ist, wird geprüft, ob die aktuelle Spalte schon gleich INDX beträgt. Ist dies der Fall, wird kein Zeichen mehr geholt und das Bit 7 von CRSW wird zur Kennzeichnung des Eingabeendes gesetzt. Ein Versuch, trotz gesetzten Bits 7 in CRSW weiter Zeichen zu holen, wird beim ersten Mal damit beantwortet, daß CRSW gelöscht und ein Carriage Return auf den Bildschirm ausgegeben wird. Der nächste Aufruf zum Zeichenholen führt damit zur Eingabeschleife der weiter unten beschriebenen Eingabeart. Ist das aktuelle Ausgabegerät gerade nicht der Bildschirm, wird einfach das RETURN als gelesener Wert übergeben.

3.2.5.2 Eingaben mit blinkendem Cursor

Hier werden zunächst Eingaben von der Tastatur geholt und auf den Bildschirm ausgegeben. Wir erinnern uns, daß die Zeichenausgabe das CRSW-Flag löscht. Bei dieser Art der Eingabe ist CRSW also gleich Null. Gestartet wird die Eingabe gewöhnlich durch den Aufruf der Kernal-Routine BASIN mit der Tastatur als Eingabegerät, was so nicht ganz richtig ist, denn die endgültigen Zeichen, die BASIN zurückliefert, stammen nicht direkt von der Tastatur, sondern vom Bildschirm.
Zur Ausgabe auf den Bildschirm gehört auch das Bewegen des Cursors und die Ausführung der anderen Steuerbefehle. Dies ist ja nichts weiter als die Ausgabe eines bestimmten ASCII-Wertes auf den Bildschirm.
Das Lesen der Zeichen von der Tastatur und die anschließende Ausgabe auf den Bildschirm geht solange voran, bis ein Carriage Return eingetippt wurde. Durch dieses

RETURN wird die Schleife verlassen. Sie liegt übrigens bei $c258 und hat auch einen eigenen Namen bekommen, weil sie so häufig benutzt wird. Sie heißt LOOP4. In dieser Schleife wird auch das Blinken des Cursors veranlaßt, außer, daß von der Tastatur gelesen und auf den Schirm ausgegeben wird. Wie dies geschieht, kann im nachfolgenden Unterkapitel nachgelesen werden.
Nach dem Erkennen des RETURN wird das eigentliche Holen von Zeichen vorbereitet. Es findet in der gleichen Art statt, wie es im vorherigen Abschnitt beschrieben wurde. Nur die Eingabegrenzen werden erst noch festgesetzt:
Das CRSW-Flag wird ungleich Null gemacht, damit eine Eingabe direkt vom Bildschirm möglich wird. Das Blinken des Cursors wird abgeschaltet.
Anhand der Zeilenüberlauf-Bits wird jetzt getestet, ob die aktuelle Cursor-Zeile noch Folgezeilen besitzt, ob also die gesamte Eingabezeile über mehrere Bildschirmzeilen verteilt ist. Wenn es noch Folgezeilen gibt, wird die Cursor-Zeile erhöht, bis die letzte Bildschirmzeile der Eingabezeile gefunden ist.
Der nächste Schritt besteht darin, unnötige Leerzeichen am Ende der Eingabezeile zu eliminieren. Solche Leerzeichen sollen beim späteren Holen der Zeichen vom Bildschirm ignoriert werden. Dazu werden über den Pointer PNT die Zeichen der Eingabezeilen von hinten her gelesen, bis ein Zeichen ungleich Blank gefunden ist.
In diesem Moment ist aber schon alles klar für eine Eingabe von Zeichen nach dem Muster, wie es in 3.2.5.1 beschrieben ist: Das Ende der Eingabezeile kommt nach LINTMP, die letzte Eingabespalte nach INDX und es könnte losgehen, wenn auch noch der Start der Eingabezeile bekannt wäre.
Dieser aber wird in den Zeropage-Adressen

LSXP = $e8 Startspalte für Eingabe

und

LSTP = $e9 Startzeile für Eingabe

vorausgesetzt. Wird die Eingabe durch die Routine BASIN eingeleitet, werden die beiden Zeiger durch BASIN gesetzt. Durch die Angabe der beiden Startwerte wird vermieden, daß ein vorher ausgegebener Text, zum Beispiel die Dialogmeldung eines Input-Befehls, bei der anschließenden Eingabe mitgelesen wird.
Als letztes bleibt dann nur noch der Übergang zur Routine, die, wie vorher beschrieben, direkt vom Bildschirm liest.

3.2.6 Das Blinken des Cursors

Aufgrund einer erstaunlichen Anzahl von Fragen, wie das Cursor-Blinken beim C128 gesteuert werden kann, möchte ich auch dieses Kapitel noch kurz anreißen.

Das eigentliche Blinken, das An- und Abschalten des Cursors, wird während des Interrupts hervorgerufen, beziehungsweise beim VDC selbständig von der Hardware erzeugt. Das VDC-Blinken kann im Register 10 direkt beeinflußt werden, das VIC-Blinken wird durch insgesamt 5 Zähler und Schalter gesteuert:

BLNON	= **$a26**	schaltet den Cursor auf starr oder blinkend (0 = blinkend, $40 = starr und angeschaltet, $c0 = starr und aus)
BLNSW	= **$a27**	schaltet den Cursor ganz aus (ungleich Null)
BLNCT	= **$a28**	Zähler für die Blinkfrequenz. BLNCT kann nicht beeinflußt werden, da der Zähler während des Interrupts initialisiert wird
GDBLN	= **$a29**	Zeichen vor dem Blinken unter dem Cursor
GDCOL	= **$a2a**	Farbe unter dem Cursor vor dem Blinken

Auch für den Blinkmodus des VDC hat das Kernal eine eigene Speicherstelle:

CURMOD	= **$a2b**	Hier wird der Cursor-Modus des VDC in der gleichen Art aufbewahrt wie er im Register 10 des VDC zu finden ist.

Da der VDC den Austausch der Schriftfarbe und der Hintergrundfarbe nicht selbständig ausführt, wird dieser Austausch durch die Kernal-Routinen vorgenommen. Die jeweilige Wechselfarbe steht dann in

CURCOL = **$a35**

Dieses etwas komplizierte Ein- und Ausschalten des Cursors muß man glücklicherweise nicht selbst übernehmen. Man kann es zwei Routinen des Kernal-Editors überlassen:

CRSON = **$cd6f (52591)**

und

CRSOFF = **$cd9f (52639)**

Die beiden Unterprogramme regeln das Ein- und Ausschalten des Cursors für beide Bildschirme, je nach Zustand von MODE ($d7).
Anhand eines kleinen BASIC-Programms kann man die Wirkung einmal nachprüfen. Es soll mit get ein Zeichen von der Tastatur gelesen werden, wobei der Cursor sichtbar sein soll:

```
10 sys dec("cd6f")
20 get a$ : if a$="" then 20
30 sys dec("cd9f")
```

Wie man sieht, blinkt der Cursor. Das Abschalten des Cursors mit CRSOFF ist notwendig, wenn man vermeiden will, daß der gesetzte Cursor eventuell auf dem Bildschirm stehenbleibt.

3.3 Der IEC-Bus

Die meisten Geräte, die an den C128 angeschlossen werden können, werden mit dessen IEC-Bus verbunden. Dieser Commodore-spezifische Bus dient der seriellen Datenübertragung zwischen verschiedenen Geräten, die gleichzeitig an die Bus-Leitungen angeschlossen sind.

3.3.1 Aufbau des IEC-Bus

Der IEC-Bus besteht von der Hardware her aus sechs Leitungen, von denen allerdings nur zwei beziehungsweise drei direkt der Datenübertragung dienen:

RESET	leitet einen Reset-Impuls vom Computer zu den am IEC-Bus angeschlossenen Peripheriegeräten, die dadurch beim Einschalten oder bei einem RESET des Computers ebenfalls in den Einschaltzustand versetzt werden.
GROUND	ist eine gemeinsame Masseleitung.
DATA	die Datenleitung des IEC-Bus.
CLOCK	die Taktleitung des langsamen Übertragungsmodus.
SRQ	hat im normalen langsamen Übertragungsmodus keine Funktion. Bei der schnellen seriellen Übertragung dient diese Leitung anstelle von CLOCK als Taktleitung.
ATN	(Attention). Um normale Datenbytes von Sendungen zu unterscheiden, die eine Anweisung an ein Peripheriegerät enthalten, wird die Attention-Leitung während der Übertragung einer Anweisung aktiviert. Anweisungen können zum Beispiel das Senden der Geräte- oder der Sekundäradresse sein.

Wie man aus der Beschreibung der einzelnen Leitungen des IEC-Bus schon erkennen kann, muß man zwei verschiedene Übertragungsmodi unterscheiden – die normale (langsame) Übertragung, wie sie zum Beispiel zwischen einer 1541-Floppy und dem C128 stattfinden wird, und die schnelle Übertragung, die nur mit den neuen Floppies 1570 und 1571 möglich ist.

Die neue schnelle Übertragungsart wurde dadurch möglich, daß die bis dato unbenutzte Leitung SRQ des IEC-Bus als Taktleitung für den sogenannten FAST-SERIAL-MODE hergenommen wurde.
Dadurch bedingt, daß diese neue Taktleitung von anderen Geräten als den neuen Floppies gar nicht wahrgenommen werden kann, ist die schnelle serielle Übertragungsweise für andere Geräte praktisch transparent. Eine 1541-Floppy bemerkt nicht einmal, daß der C128 sie »schnell« anzusprechen versucht.
Der Geschwindigkeitsvorteil der schnellen Übertragung wird dadurch erreicht, daß die Daten nicht mehr über das PRA des CIA 2, sondern über das serielle Port-Register des CIA 1 übertragen werden. Die zur Übertragung notwendigen Taktimpulse werden nicht mehr softwaremäßig durch das Beschreiben und Lesen des PRA des CIA 2 gewonnen, sondern hardwaremäßig durch den Einsatz des Timers A des CIA 1, indem er Daten in das serielle Port-Register hinein- beziehungsweise aus dem Register hinaustaktet.
Alle Datensendungen, die gleichzeitig mit einem ATN-Impuls übertragen werden müssen, werden weiterhin über den langsamen Modus gesendet, damit alle an den Bus angeschlossenen Geräte sie aufnehmen können.
Kontrolliert wird der schnelle serielle Modus durch das FSDIR-Bit des Mode-Konfigurationsregisters der MMU (Bit 3, $d505). Ist es gesetzt, ist eine schnelle Übertragung möglich, ist es gelöscht, kann nur im langsamen Modus übertragen werden.
Meiner Ansicht nach schaltet FSDIR die beiden Leitungen SP (Ein-/Ausgang des seriellen Datenregisters der CIA 1) und CNT (Ein-/Ausgang des Timers A) zum IEC-Bus hinzu. Das Handbuch des C128 schweigt sich aber zu der genauen Rolle des FSDIR-Bit wieder einmal aus, so daß die Rolle dieses Bits wohl erst durch den Einsatz eines Oszillographen geklärt werden wird.
Der Geschwindigkeitszuwachs durch die schnelle serielle Übertragung – getestet mit einer 1571 – beträgt bei Verwendung einer im 1541-Format formatierten Diskette etwa:

```
LOAD   Faktor 5
SAVE   Faktor 1.5
get #  Faktor 2
```

Die größere Geschwindigkeit beim Laden von Programmen wird durch das später noch beschriebene besondere Ladeverfahren im BURST-MODE erreicht, das von der 1571-Floppy besonders unterstützt wird.
Noch höhere Geschwindigkeiten werden erreicht, wenn die bearbeitete Diskette im 1571-Format formatiert wurde – erkennbar durch die Meldung, daß über 1300 Blöcke frei sind:

```
LOAD   ca. 10
SAVE   ca. 1.5
get #  ca. 2.7
```

Geräte- und Sekundäradressen

Wie beschrieben, ist der IEC-Bus dadurch gekennzeichnet, daß alle angeschlossenen Geräte gleichzeitig mit dem Bus verbunden sind. Diese Struktur macht es notwendig, den einzelnen Geräten Geräteadressen zuzuweisen, anhand derer sie erkennen können, ob sie angesprochen sind oder nicht. Die Geräteadresse wird mit der Ausgabe des ATN-Signals auf den IEC-Bus gesendet.

Die Geräteadresse besteht aus einem Byte, unterteilt in eine obere und untere Hälfte, die eine unterschiedliche Bedeutung haben.
Die untere Hälfte (Bits 4 bis 0) stellt die angewählte Gerätenummer dar, die der Gerätenummer entspricht, die von BASIC her bekannt sein dürfte.
Die obere Hälfte (Bits 7 bis 5) enthält die Information, zu welchem Zweck das Peripheriegerät angesprochen wird.
Ein Senden der Gerätenummer ist nur dann erforderlich, wenn ein Gerät das erste Mal angesprochen wird.

Folgende Geräteadressen sind möglich:

Geräteadresse	Bedeutung
% 001x xxxx	Das Gerät soll Daten empfangen. Es soll zuhören (LISTEN)
% 010x xxxx	Das Gerät soll Daten senden. Es soll sprechen (TALK)
% 0011 1111	Das Gerät soll aufhören, Daten zu empfangen (UNLISTEN)
% 0101 1111	Das Gerät soll aufhören, Daten zu senden (UNTALK)

Nach dem Senden der Geräteadresse weiß das angesprochene Gerät schon einmal, daß es angesprochen ist und was es im großen und ganzen zu tun hat. Da die meisten Geräte aber mehr können als nur stur zu senden oder zu empfangen (die Floppy kann zum Beispiel mehrere Files gleichzeitig offenhalten) und Daten aus diesen Files senden oder empfangen, ist es sinnvoll, daß nach der Geräteadresse ein weiteres Byte auf den IEC-Bus ausgegeben wird, die Sekundäradresse, die dem Gerät weitere Einzelheiten mitteilt. Im Fall der Floppy zum Beispiel, um welches File es sich handeln soll.

Auch die Sekundäradresse zerfällt in die eigentliche Sekundäradresse und einen Teil, in dem mitgeteilt wird, zu welchem Zweck die Sekundäradresse gesendet wird. Dieser Zweck läßt sich wieder in einer Tabelle festhalten:

Sekundäradresse	Bedeutung
% 011y yyyy	Normale Sekundäradresse
% 1111 yyyy	Öffnen eines Files innerhalb der Floppy (OPEN)
% 1110 yyyy	Schließen eines Files innerhalb der Floppy (CLOSE)

Die normale Sekundäradresse gilt für alle Geräte. Beim Umgang mit der Floppy sind nur die Sekundäradressen bis 15 erlaubt. Dafür hat man das Bit 4 der Sekundäradresse der OPEN- beziehungsweise CLOSE-Funktion zugeschlagen.
Dieses Open und Close ist nicht zu verwechseln mit dem Open und Close, das Computer-intern vorgenommen wird!
Die Open- beziehungsweise Close-Sekundäradresse wird vielmehr nur dann benutzt, wenn File-Namen übertragen werden müssen, wenn also Files mit Name auf der Floppy geöffnet oder geschlossen werden sollen.
Dazu wird hinter der Open-Sekundäradresse einfach der File-Name zeichenweise auf den IEC-Bus ausgegeben. Beendet wird das Senden des Namens durch den UNLISTEN-Befehl:

```
LISTEN
Sekundäradresse für OPEN
Name
UNLISTEN
```

Ob eine schnelle serielle Übertragung möglich ist, wird innerhalb des Betriebssystems selbständig geprüft, indem beim Senden von LISTEN oder TALK ein Datenbyte über das serielle Port-Register »schnell« ausgegeben und eine Antwort des Laufwerks im schnellen Modus abgewartet wird. Dieses Verfahren wird von Geräten, die keinen schnellen Modus besitzen, nicht bemerkt, da die normale Taktleitung CLOCK nicht benutzt wird.

Die schnelle Floppy wird durch die Antwort von

% 1111 1111 = Host Requests Fast

auf dem Bus erkannt. Der Computer teilt seinerseits mit

% 0000 0000 = Device Requests Fast

mit, daß er zur schnellen Übertragung bereit ist.

Das Ergebnis der Abfrage wird vom Betriebssystem in der Speicherstelle

SERIAL = $a1c

temporär festgehalten, so daß bei der folgenden Ausgabe von Daten beziehungsweise beim Empfang der Daten die schnellen oder die langsamen Bus-Routinen angesprungen werden können. Bit 6 und 7 des SERIAL-Flags sind gesetzt, wenn der schnelle serielle Modus möglich ist.

Das Statusbyte (STATUS, $90)

Der IEC-Status müßte eigentlich von BASIC aus bekannt sein. Er entspricht exakt der BASIC-Statusvariablen ST. Wie schon beim C64 ist der IEC-Status in der Zeropage bei

STATUS = $90

zu finden. Das gleiche Byte dient auch zur Aufbewahrung des Kassetten-Status.

Das Statusbyte entspricht dem, was der Computer bei IEC-Bus-Aktionen als Echo vom IEC-Bus aufnimmt. So kann zum Beispiel die Floppy zusammen mit einem Datenbyte die Rückmeldung liefern, daß ein File vollständig übertragen ist, oder es kann sich herausstellen, daß ein adressiertes Gerät auf dem IEC-Bus keine Antwort gibt oder ähnliches. Insgesamt werden vier solche Fehlerfälle oder Meldungen in STATUS festgehalten, so daß der Programmierer darauf reagieren kann:

Bit 7 = 1	bedeutet, daß ein adressiertes Gerät keine Antwort gibt. Dieser Fall entspricht einem »device not present«.
Bit 6 = 1	Dieses Bit kennzeichnet das Ende einer Übertragung. Die Floppy liefert dieses Bit, das End-of-File-Bit (EOF) zusammen mit dem letzten Datenbit.
Bit 2 = 1	Ein Lesefehler ist aufgetreten. Nach der Adressierung eines Gerätes mit TALK sendet dieses Gerät nicht rechtzeitig ein Datenbyte.
Bit 1 = 1	Ein Schreibfehler, der daran erkannt wird, daß das empfangende Gerät den Erhalt eines Datenbytes nicht rechtzeitig mit einem Quittungsimpuls bestätigt.

Das Löschen des STATUS-Bytes erfolgt nicht automatisch, deshalb ist es unbedingt erforderlich, vor Beginn einer Datenübertragung in Maschinensprache das STATUS-Byte mit Null zu initialisieren!

3.4 Der Umgang des Betriebssystems mit den Peripheriegeräten

In diesem Abschnitt soll in der Hauptsache beschrieben werden, wie mit einer angeschlossenen Floppy von Maschinensprache aus gearbeitet wird. Alle Beispiele und Erläuterungen gelten aber sinngemäß auch für andere Geräte, die den IEC-Bus benutzen.

Auf eine genauere Beschreibung der RS232-Behandlung und der Kassetten-Routinen des Kernals habe ich verzichtet, da beide gegenüber den bisher beschriebenen Ein-/Ausgabegeräten nur eine untergeordnete Rolle spielen.

3.4.1 Das direkte Ansprechen des IEC-Bus

Neben einer später beschriebenen Methode, die eine bequemere Behandlung mehrerer gleichzeitig oder quasi-gleichzeitig benutzter Geräte erlaubt, existiert auch ein direkter Weg, ein Gerät auf dem IEC-Bus anzusprechen. Zur Unterstützung dieser Möglichkeit bietet das Kernal acht Routinen an, die alle über die Sprungleiste am oberen Ende des Kernal-ROMs zu erreichen sind. Mit dieser Sprungleiste hat es eine besondere Bewandtnis, denn man kann sich darauf verlassen, daß alle Routinen, die in der Sprungleiste aufgeführt sind, bei etwaigen ROM-Änderungen ihren Platz in der Leiste nicht verändern. Will man sicher sein, daß ein selbstentwickeltes Programm auch auf späteren Versionen des C128 noch läuft, sollte man deshalb darauf achten, daß man möglichst nur Routinen der Sprungleiste $ff47 bis $ffff verwendet.
Die acht in unserem Zusammenhang interessierenden Routinen zur Ein- und Ausgabe vom beziehungsweise auf den IEC-Bus sind:

LISTEN	= **$ffb1**	sendet LISTEN an ein Gerät
SECLST	= **$ff93**	sendet die Sekundäradresse für LISTEN
UNLIST	= **$ffae**	sendet UNLISTEN
TALK	= **$ffb4**	sendet TALK
SECTLK	= **$ff96**	sendet die Sekundäradresse für TALK
UNTALK	= **$ffab**	sendet UNTALK
IECIN	= **$ffa5**	holt ein Byte vom IEC-Bus
IECOUT	= **$ffa8**	sendet ein Byte auf den Bus

Mit diesen Routinen kann man eigentlich schon alles machen, was man auf dem IEC-Bus unternehmen will. Die Abfrage auf den schnellen seriellen Modus wird durch LISTEN und TALK schon durchgeführt, wir brauchen uns also selbst nicht darum zu kümmern, welche Floppy angeschlossen ist.
Sehen wir uns erst einmal ein paar Beispiele an, wie die Routinen bedient werden.
Als erstes könnte man zum Beispiel einen Befehl an die Floppy senden, ein Standardbeispiel. Die Floppy soll im Beispiel zuhören, sie muß also als LISTENER adressiert werden. Das macht die LISTEN-Routine für uns, die mit der Gerätenummer im Akku aufgerufen wird. Die Sekundäradresse muß 15 sein, da die Floppy nur über diese Sekundäradresse Befehle annimmt. Das macht SECLST, allerdings müssen wir die oberen Bits der Sekundäradresse selbst richtig setzen. In der Tabelle sehen wir, daß es sich weder um ein Öffnen noch um das Schließen einer Datei handelt, die oberen Bits sind also %011. Mit anderen

Worten gesagt, wir müssen zur eigentlichen Sekundäradresse 15 noch $60 = 96 = %0110 0000 addieren, damit wir das als Sekundäradresse zu sendende Byte erhalten. Nachdem das alles vorbereitet ist, können wir mit IECOUT unseren Befehl übermitteln. Am Schluß unserer kleinen Routine fehlt dann nur noch, der Floppy mitzuteilen, daß die Sendung zu Ende ist, wir senden dazu UNLISTEN mit der Routine UNLIST, die keine Eingabewerte benötigt. Damit steht unsere Routine schon.
Hier das entsprechende Listing, das der Einfachheit halber so wiedergegeben ist, wie es der ASE-Makroassembler für den 128er akzeptiert. Assemblerspezifische Befehle werden an passender Stelle im Kommentar erläutert:

Senden eines Befehls an die Floppy

```
        .base $b00              ; Startadresse im Kassettenpuffer

.define listen = $ffb1          ; Labeldefinitionen
.define seclst = $ff93
.define unlist = $ffae
.define iecout = $ffa8

start   lda #8                  ; Gerätenummer der Floppy
        jsr listen              ; LISTEN senden
        lda #15+$60             ; Sekundäradresse
        jsr seclist             ; senden
        ldx #0                  ; Startwert = 0
loop    lda text,x              ; Zeichen des Befehls
        jsr iecout              ; auf IEC-Bus senden
        inx                     ; Index erhöhen
        cpx #2                  ; war letztes Zeichen?
        bne loop                ; nein
        jmp unlist              ; UNLISTEN senden und rts
text    .byte "i0"              ; auszugebender Befehl
```

Wer es ausprobiert hat, wird festgestellt haben, daß es funktioniert. Eines stört den gewissenhaften Programmierer aber noch an der Routine: Wo und wie werden eventuelle Fehler abgefangen, zum Beispiel wenn gar keine Floppy angeschlossen ist?
Um eine größere Übersichtlichkeit der Beispiele zu erreichen, sind in ihnen keine Fehlerabfragen enthalten. Diesem Thema ist aber weiter unten ein eigener Abschnitt gewidmet, so daß es keine Schwierigkeiten machen dürfte, die Beispiele um eine eigene Fehlerabfrage zu erweitern.
Ganz analog zum Senden eines Befehls an die Floppy kann man natürlich auch den Fehlerkanal der Floppy abfragen. Die Überlegungen sind dabei im Prinzip dieselben, die schon

dem Senden zugrunde lagen. Hinzu kommt, daß im Statusbyte das Ende der Übertragung angezeigt wird. Ein Listing hierzu:

```
start   lda #0          ; Statusbyte löschen
        sta status
        lda #8          ; Floppy als TALKER
        jsr talk        ; adressieren
        lda #15+$60     ; Kanal 15 und normale Sek.-Adr.
        jsr sectlk      ; als Sekundäradresse
loop    jsr iecin       ; Byte vom Bus holen
        jsr print       ; auf Bildschirm ausgeben
        bit status      ; Bit 6 des STATUS = EOF
        bvc loop        ; nicht gesetzt: weiter
        jmp untalk      ; gesetzt: UNTALK und rts
```

Eigentlich eine vollkommen einfache Sache, nicht wahr?

Es bleibt aber noch eine Frage kurz zu klären: Warum existieren unterschiedliche Routinen zum Senden der Sekundäradresse bei LISTEN und TALK, wo doch genau das gleiche zu tun ist? Die Antwort ist wie in den meisten Fällen recht einfach. Der Unterschied der beiden Routinen liegt darin begründet, daß der Computer einmal Daten sendet, das andere Mal Daten empfängt. Beim Datenempfang kann aber der Fall eintreten, daß schon beim Senden der Sekundäradresse über den Bus zurückgemeldet wird, daß ein File vollständig gelesen ist. Dieses EOF wird in der SECTLK-Routine erkannt und entsprechend behandelt. Mehr dazu bei der Beschreibung der I/O-Fehlerbehandlung.

Als letztes Beispiel für das direkte Arbeiten mit dem IEC-Bus möchte ich ein Schreib-File vom Typ PRG auf einer Diskette eröffnen. Wir erinnern uns vielleicht, daß die Kanalnummern 0 und 1 der Floppy für die Behandlung von PRG-Files reserviert sind. Eine Nummer von 0 bedeutet, daß ein PRG-File als Lesedatei geöffnet wird, eine Nummer von 1 öffnet das File als Schreibdatei.

Hier die Öffnungsprozedur, die für unser Beispiel anzuwenden ist:

```
open    lda #8          ; Floppy als LISTENER
        jsr listen      ; adressieren
        lda #1+$f0      ; Kanalnummer + OPEN-Zusatz
        jsr seclst      ; als Sekundäradresse
        ldx #0          ; Schleifenzähler
loop    lda name,x      ; Zeichen des Namens
        jsr iecout      ; auf den Bus senden
```

```
        inx             ; alle Zeichen?
        cpx #4
        bne loop        ; nein: weiter
        jmp unlist      ; ja: UNLISTEN und rts

name    .byte "test"    ; File-Name für das Beispiel
```

Das Schließen der offenen Datei erfolgt ganz entsprechend:

```
close   lda #8          ; Floppy als LISTENER
        jsr listen
        lda #1+$e0      ; Kanalnummer + CLOSE-Zusatz
        jsr seclst      ; als Sekundäradresse
        jmp unlist      ; UNLISTEN und rts
```

Das Öffnen und Schließen anderer Dateitypen erfolgt ganz analog, nur wird man hier nicht ohne die Zusätze ,s,r oder ,u,w zum File-Namen auskommen, um den Unterschied zwischen einer Lese- und einer Schreibdatei für die Floppy erkennbar zu machen.
Bevor wir zur Besprechung der Bus-Fehlerbehandlung kommen, möchte ich einige Routinen des Kernals vorstellen, die die Arbeit mit den Peripheriegeräten erheblich erleichtern. Wie man schon an den obigen Beispielprogrammen sieht, ist es doch eine ganz schöne Arbeit, selbst eine so einfache Aktion wie ein CLOSE auf dem Bus durchzuführen, wenn man alles direkt programmieren muß.

3.4.2 Das Arbeiten mit logischen File-Nummern

Das Konzept der logischen File-Nummern besteht darin, daß dem stets benötigten Adressierungspaar Geräteadresse/Sekundäradresse ein Index, also eine Nummer, die logische File-Nummer, zugewiesen wird. Unter dieser Nummer bewahrt das Betriebssystem die beiden Adressierungen auf, so daß sie durch Angabe der logischen File-Nummer bei Bedarf von den entsprechenden I/O-Routinen nachgeschlagen werden können. Einträge in die Tabellen müssen sich nicht notwendigerweise auf Geräte am IEC-Bus beziehen. Genauso können auch der Bildschirm (Gerät 3), die Tastatur (Gerät 0) oder die RS232 angesprochen werden.
Insgesamt kann sich das Kernal 10 solcher Einträge auf einmal merken. Die einzelnen Einträge werden in den Tabellen

LFNTAB	**= $362**	Tabelle der logischen File-Nummern
GATAB	**= $36c**	Geräteadressen
SATAB	**= $376**	Sekundäradressen

aufbewahrt. Die Sekundäradresse wird bei der Aufnahme in die Tabelle der Sekundäradressen durch die OPEN-Routine automatisch mit dem normalen Zusatz versehen. Im Fall der Floppy bedeutet dies, daß der Eintrag in die Sekundäradressen-Tabelle aus der Kanalnummer + $60 besteht.
Die Anzahl der Tabelleneinträge, aus einem etwas später einsichtigen Grund auch Anzahl der offenen Files genannt, ist in der Zeropage-Adresse

LDTND = $98

nachzulesen.

Das gerade aktuell angesprochene Ein- und Ausgabegerät ist in den Speicherzellen

DFLTN = $99 Eingabegerät

und

DFLTO = $9a Ausgabegerät

aufbewahrt. Je nach eingestelltem Gerät verzweigen die allgemeinen Ein-/Ausgaberoutinen BASIN und BSOUT zu den verschiedenen gerätespezifischen Unterroutinen. Im Fall des IEC-Bus sind dies die vorher beschriebenen IECIN und IECOUT.

3.4.2.1 Einträge einfügen und löschen (OPEN/CLOSE)

Die beiden Kernal-Routinen OPEN und CLOSE sind zuständig für das Einfügen und Löschen von Tabelleneinträgen. Gleichzeitig eröffnet die OPEN-Routine das entsprechende File in derselben Weise, wie es auch durch die vorher geschilderte direkte Programmierung erfolgen könnte. CLOSE schließt genauso das File wieder. Aus diesem Grund ist die Zahl der Tabelleneinträge gleich der Zahl offener Files. Die beiden Routinen liegen bei

OPEN	**= $ffc0**
CLOSE	**= $ffc3**

Um mit OPEN ein File eröffnen zu können, ist es selbstverständlich erforderlich, die notwendigen Parameter in irgendeiner Weise an die Routine zu übergeben. Dies geschieht in den folgenden Zeropage-Adressen:

LA	**= $b8**	Logische File-Nummer
SA	**= $b9**	Sekundäradresse
FA	**= $ba**	Geräteadresse
FNLEN	**= $b7**	Länge des File-Namens
FNADR	**= $bb/ $bc**	Adresse des File-Namens
FNBNK	**= $c7**	Bank, in der sich der File-Name befindet

Im Fall eines LOAD- oder SAVE-Aufrufes (siehe unten) kommt noch eine Speicherstelle hinzu:

BA	= **$c6**	Bank, in die geladen, oder aus der heraus abgespeichert wird

Diese ganzen Adressen kann man entweder selbst vor einem OPEN-Aufruf mit den gewünschten Werten laden, man kann dies aber auch auf viel bequemere Art einigen dafür geschaffenen Kernal-Routinen überlassen:

SETPAR	= **$ffba**	setzt logische File-Nummer, Sekundär- und Geräteadresse
SETNAM	= **$ffbd**	setzt Namenlänge und Adresse
SETBNK	= **$ff68**	setzt Bank des File-Namens und Bank für LOAD/ SAVE/ VERIFY

Der Aufruf dieser Routinen erfolgt so:

```
SETPAR  lda #1          ; logische File-Nummer
        ldx #8          ; Gerätenummer
        ldy #15         ; Sekundäradresse (Kanalnummer)
        jsr setpar

SETNAM  lda #5          ; File-Namenlänge
        ldx # <(adr)    ; Lowbyte der Namenadresse
        ldy # >(adr)    ; Highbyte der Adresse
        jsr setnam

SETBNK  lda #0          ; RAM-Bank-Nummer für Load etc.
        ldx #1          ; RAM-Bank-Nummer des File-Namens
        jsr setbnk
```

Unser Beispiel, in dem wir ein Schreib-File auf Diskette eröffnet haben, sieht unter Verwendung der obigen Routinen so aus:

```
start  ldx #0          ; Name ist in Bank 0
       jsr setbnk
       lda #1          ; logische File-Nummer soll 1 sein
       ldx #8          ; Gerät 8 = Floppy
       ldy #1          ; Kanal 1 für PRG-Typ/ Schreib-File
       jsr setpar
```

```
        lda #4          ; Namenlänge = 4
        ldx # <(name)   ; Lowbyte der Adresse
        ldy # >(name)   ; Highbyte
        jsr setnam
        jmp open        ; File öffnen und in Tabelle
                        ; eintragen/ rts
```

Das Schließen des Files mit Löschen des Tabelleneintrags erfolgt ohne Parameterübergabe in den Zeropage-Adressen. Hierzu ist nur die Übergabe der logischen File-Nummer im Akkumulator notwendig:

```
start   lda #1          ; logische File-Nummer
        jmp close       ; File schließen und Eintrag
                        ; löschen/ rts
```

3.4.2.2 Tabelleneinträge suchen/Umschaltung der Ein- und Ausgabe auf geöffnete Files (CHKIN, CKOUT, CLRCH)

Nachdem ein File erst einmal geöffnet ist, möchte man natürlich auch einmal Daten senden oder empfangen können. Bei der direkten Programmierung wurde dies durch LISTEN + SECLST beziehungsweise TALK + SECTLK ermöglicht. Verwendet man logische File-Nummern, muß man die Entsprechungen

	CHKIN	**= $ffc6**	legt die Eingabe auf ein Gerät
und			
	CKOUT	**= $ffc9**	legt die Ausgabe auf ein Gerät

verwenden. Den Routinen UNLIST beziehungsweise UNTALK entspricht

CLRCH	**= $ffcc**	Ein- und Ausgabe auf Normalwert (Tastatur und Bildschirm)

Die Änderung des aktuellen Ein- und Ausgabegerätes läßt sich in den schon erwähnten Zeropage-Adressen DFLTN und DFLTO ablesen.

Der Aufruf der drei Routinen ist sehr einfach. CLRCH benötigt keine Parameter, den beiden anderen Routinen wird einfach die logische File-Nummer im X-Register übergeben:

```
ldx #1           ; logische File-Nummer
jsr chkin        ; Eingabe vom Gerät, das unter der
                 ; logischen File-Nummer definiert ist

ldx #1           ; entsprechend für die Ausgabe
jsr ckout

jsr clrch        ; I/O auf Default (Tast./Bildsch.)
```

Die Routinen CHKIN, CKOUT, CLRCH usw. benutzen zwei Unterroutinen, die Einträge in den Tabellen suchen:

LKUPLA	= **$ff59**	sucht nach einer logischen File-Nummer
LKUPSA	= **$ff5c**	sucht nach einer Sekundäradresse

Der Aufruf dieser Routinen:

```
LKUPLA  lda #...       ; logische File-Nummer im Akku
        jsr lkupla     ; Eintrag suchen

LKUPSA  ldy #...       ; Sekundäradresse in Y
        jsr lkupsa     ; suchen
```

Die Routinen liefern im Carry-Flag die Information zurück, ob ein Eintrag gefunden wurde. Gesetztes Carry bedeutet »nicht gefunden«. Wurde ein entsprechender Eintrag gefunden, befinden sich beim rts in den Prozessorregistern die drei Tabelleneinträge in der gleichen Anordnung, wie die SETPAR-Routine sie verlangen würde. Gleichzeitig sind diese Werte auch in die zugehörigen Zeropage-Adressen geschrieben worden. Ein wichtiger Nebeneffekt: Das Löschen des STATUS-Bytes wird durch die beiden Routinen ebenfalls vorgenommen, eine weitere Entlastung des Programmierers.

Das Suchen eines Eintrags in den Tabellen kann dann sehr nützlich sein, wenn ein neuer Kanal geöffnet werden soll, aber nicht bekannt ist, welche logischen File-Nummern schon in Gebrauch sind.

Im folgenden sind noch einmal alle Routinen aufgeführt, bei denen eine automatische Löschung des STATUS-Bytes vorgenommen wird:

LKUPLA **LKUPSA** **CHKIN** **CKOUT** **OPEN**

3.4.2.3 Die Ein- und Ausgaberoutinen bei Verwendung logischer File-Nummern

Den beiden Routinen IECIN und IECOUT entsprechen bei der Verwendung der logischen File-Nummern die schon mehrfach erwähnten Routinen BASIN für die Eingabe und BSOUT für die Ausgabe.

BASIN = $ffcf
BSOUT = $ffd2

In ihnen wird je nach den in DFLTN und DFLTO eingestellten Ein-/Ausgabegeräten zu passenden Unterroutinen verzweigt. Ist zum Beispiel die Floppy als Gerät gewählt, wird zu IECIN oder IECOUT verzweigt, oder es wird nach PRINT verzweigt, wenn das Ausgabegerät der Bildschirm ist.
Bildschirm und Tastatur spielen eine besondere Rolle bei der Verwendung logischer File-Nummern. Diese beiden Geräte sind als Default-Geräte anzusehen, das heißt, immer wenn nichts anderes ausdrücklich vereinbart worden ist, gilt die Tastatur als Ein- und der Bildschirm als Ausgabegerät. Man muß diese Geräte also nicht extra über OPEN ansprechen.

3.4.3 Laden, Verifizieren, Abspeichern, BOOT

Auch die Routinen zum Laden und Abspeichern von Files sind Bestandteil des Betriebssystems. Wie schon bei BASIN und BSOUT wird auch hier in Abhängigkeit vom gewählten Gerät in Unterroutinen verzweigt. Zur Verfügung stehen Routinen zur Bedienung des Datenrecorders und der Floppy.
Das Booten, eine Spezialform des Ladens mit einem Autostart des geladenen Programms, kann nur von Diskette erfolgen.

3.4.3.1 Laden und Verifizieren (LOAD, VERIFY)

Beide Aktionen werden von einer Kernal-Routine abgehandelt:

LOAD = $ffd5

Ein im Akkumulator übergebener Schalter entscheidet, ob ein LOAD oder ein VERIFY durchgeführt wird:

Akku	
0	LOAD
1	VERIFY

Der Zustand des Schalters wird im Load-/Verify-Flag festgehalten:

VERCK = $93

Alle File-Parameter wie File-Name, gewähltes Gerät usw. werden durch die Routinen SETPAR, SETNAM und SETBNK gesetzt. Die Angabe einer logischen File-Nummer entfällt natürlich. Die Sekundäradresse muß wie folgt gewählt sein:

Sek.Adr.	
0	für relatives Laden wie beim BASIC-DLOAD, die Ladeadresse wird beim LOAD-Aufruf übergeben. Die abgespeicherte Startadresse wird ignoriert.
nicht 0	für absolutes Laden wie beim BASIC-BLOAD. Es wird an die Adresse geladen, die im File angegeben ist.

Die Festlegung der RAM-Bank, in die geladen wird, erfolgt über die SETBNK-Routine (siehe dort). Die Angabe der Startadresse beim relativen Laden geschieht im X- und Y-Register.

Beispiele für relatives Laden und absolutes Verify:

```
relativ   ldx #8            ; Gerät = Floppy
          ldy #0            ; für relatives Laden
          jsr setpar
          lda #...          ; File-Namenlänge
          ldx #<(adr)       ; Lowbyte der Namenadresse
          ldy # >(adr)      ; Highbyte
          jsr setnam
          lda #0            ; RAM-Bank, in die geladen wird
          ldx #0            ; RAM-Bank, in der der Name steht
          jsr setbnk
          lda #0            ; Flag für LOAD
          ldx # <(start)    ; Lowbyte der Startadresse
          ldy # >(start)    ; Highbyte
          jsr load

absolut   ldx #8            ; Gerät
          ldy #1            ; für absolutes Laden/Verify
          jsr setpar
          lda #...          ; Namenlänge
          ldx # <(adr)      ; Low- und
          ldy # >(adr)      ; Highbyte der Namenadresse
          jsr setnam
          lda #1            ; Bank für Load/ Verify/ Save
          ldx #0            ; Bank des File-Namens
```

```
jsr setbnk
lda #1              ; Flag für Verify
jsr load
```

Zu Ende der LOAD-Routine wird im X- und Y-Register die Endadresse des Ladens oder Verifizierens zurückgeliefert. Das X-Register enthält das Low-, das Y-Register das Highbyte dieser Adresse, die hinter das letzte abgespeicherte Byte weist.

Die Adresse findet sich auch in der Zeropage wieder, in:

EAL = $ae / $af

Als Speicher für die Startadresse bei relativem Laden dient der LOAD-Routine der Zeiger

MEMUSS = $c3 / $c4

Ein Löschen des STATUS-Bytes ist vor dem LOAD-Aufruf nicht erforderlich.

Das Laden im schnellen seriellen Modus erreicht seine hohe Geschwindigkeit nicht allein durch die größere Daten-Übertragungsgeschwindigkeit. Vielmehr erzielt das schnelle LOAD, das auch LOAD im BURST MODE genannt wird, die Geschwindigkeit erst durch die Unterstützung durch eine schnelle Floppy 1570 oder 1571. Hierzu wird das Laden im BURST MODE der schnellen Floppy durch den Befehl

```
u0;chr$(31) ;File-Name
```

angezeigt. Die schnelle Floppy überträgt auf diesen Befehl hin beim anschließenden BURST-MODE-LOAD das zu ladende Programm blockweise, das heißt immer 255 Byte schnell hintereinander, wobei der Computer den Datenempfang nur über die CLOCK-Leitung quittieren muß. Der letzte Block wird durch die Floppy gekennzeichnet und seine Länge gesondert übertragen.

3.4.3.2 Abspeichern (SAVE)

Auch beim Abspeichern werden die File-Parameter durch die Routinen SETPAR etc. gesetzt. Die SAVE-Routine selbst befindet sich bei

SAVE = $ffd8

Eine Unterscheidung zwischen relativem und absolutem Speichern gibt es natürlich nicht. Bei SAVE hat deshalb weder die logische File-Nummer noch die Sekundäradresse eine Bedeutung.
Da aber immer ein Bereich abgespeichert wird, müssen der SAVE-Routine zwei Adressen, die Start- und die Endadresse des abzuspeichernden Bereichs, mitgeteilt werden. Die Endadresse wird genau wie bei LOAD im X- und Y-Register übergeben. Sie muß auf das erste Byte hinter dem abzuspeichernden Bereich zeigen. Die Startadresse wird in zwei Byte in der Zeropage übergeben, die der Benutzer frei wählen kann. Die SAVE-Routine transportiert diese Startadresse in den Zeiger

STAL = $c1
STAH = $c2

Die Endadresse wird in EAL abgelegt. Die Angabe des Zeropage-Zeigers eigener Wahl erfolgt beim SAVE-Aufruf dadurch, daß das Lowbyte des Zeigers übergeben wird.

Beispiel:

```
ldx #8          ; Gerät = Floppy
jsr setpar
lda #...        ; Länge des File-Namens
ldx # <(adr)    ; Low- und
ldy # >(adr)    ; Highbyte der Adresse des Namens
jsr setnam
lda #1          ; RAM-Bank, in der der abzuspeichernde Bereich liegt
ldx #0          ; Bank des File-Namens
jsr setbnk
lda #$fd        ; Lowbyte des Zeigers $fd/$fe, der die Startadresse
                  enthalten soll
ldx # <(end)    ; Lowbyte der Endadresse
ldy # >(end)    ; Highbyte
jsr save
```

Wieviel der BURST MODE an Geschwindigkeit ausmacht, wird am Beispiel von SAVE sehr deutlich, das keinen BURST MODE kennt. Bei SAVE werden die Daten nach LISTEN und SECLST einfach über die Routine IECOUT ausgegeben.

3.4.3.3 Das Booten von Diskette

Das Booten (aus dem Englischen: in die Stiefel helfen) ist eine Art des Ladens, die es ermöglicht, ein Programm von einer Diskette in den Speicher zu holen und anschließend

auch zu starten. Ein Booten von Kassette ist nicht möglich. Beim Booten wird immer das Laufwerk mit der Gerätenummer 8 angesprochen, das vom Kernal als das Hauptlaufwerk angesehen wird.
Um den Auto-Start-Effekt zu erzielen, wird der Sektor 0 der ersten Spur der zu bootenden Diskette zu Hilfe genommen. Diesen Block liest das Betriebssystem beim Booten mit Hilfe einiger Routinen, die zusammenfassend auch das DOS (Disk Operating System) des Betriebssystems genannt werden.
Im normalen Betrieb des 128ers sind zwei Arten des Bootens möglich. Zum einen das Booten beim Einschalten des Computers beziehungsweise bei einem RESET. Dies ist das eigentliche Booten.
Zum anderen existiert auch ein BASIC-Befehl BOOT, der allerdings mit einem BOOT-Vorgang nichts zu tun hat.
Obwohl der BASIC-Befehl natürlich nicht unter das Thema Betriebssystem paßt, möchte ich ihn doch an dieser Stelle abhandeln, zumal sich das Handbuch des 128ers hartnäckig weigert, Auskunft darüber zu liefern, wie der Befehl im einzelnen arbeitet.

3.4.3.3.1 Das Booten durch den BOOT-Befehl

Beim BOOT-Befehl ist zunächst zu unterscheiden, ob er von einem File-Namen gefolgt wird oder allein steht. Folgt kein Name, wird das Booten durchgeführt, das im nächsten Abschnitt erläutert wird.
Leider funktioniert das Booten mit File-Namen nicht immer. An dieser Stelle hat nämlich das Fehlerteufelchen wieder einmal zugeschlagen, was auch die Schweigsamkeit des Handbuchs zu diesem Thema erklärt.
Planmäßig sollte der BOOT-Befehl ein Maschinenprogramm wie BLOAD in den Speicher holen und anschließend an seiner Startadresse mit JSRFAR starten. Leider haben die Programmierer der Kernal-LOAD-Routinen vergessen, diese Startadresse innerhalb der langsamen LOAD-Routine festzuhalten. Nach Plan müßte die von Diskette gelesene Startadresse in den Zeropage-Zeiger

SAL = $ac/$ad

gebracht werden, was im BURST MODE auch schön gemacht wird, beim normalen LOAD aber unterlassen wird.
Der BASIC-Befehl BOOT funktioniert deshalb nur, wenn eine 1570- oder 1571-Floppy angeschlossen ist.

3.4.3.3.2 Das Booten über den BOOT-Sektor

Das eigentliche Booten findet beim Einschalten oder bei einem RESET des Computers statt. Vom Laufwerk 0 der Floppy mit der Gerätenummer 8 wird der Sektor Nummer 0

der Spur 1 in den Kassettenpuffer ab $b00 geladen. Sind die drei ersten Zeichen des Sektors gleich »cbm«, so handelt es sich um einen BOOT-Sektor, ansonsten wird das Booten abgebrochen.

Ist ein Boot-Sektor im Puffer, wird die Meldung

```
booting
```

ausgegeben, die 4 auf das CBM folgenden Byte werden in die Zeropage übernommen, und zwar in die Zeropage-Zeiger

SAL = $ac/ $ad

und

EAL = $ae/ $af

Die vier Byte haben im einzelnen folgende Bedeutung:

$ac/ $ad	Adresse, an die die folgenden BOOT-Sektoren geladen werden sollen.
$ae	Konfigurationsindex, unter dem die folgenden BOOT-Sektoren abgespeichert werden sollen.
$af	Anzahl der noch folgenden BOOT-Sektoren.

Die folgenden Bytes innerhalb des Kassettenpuffers werden über die BSOUT-Routine auf den Bildschirm ausgegeben, bis ein Nullbyte gefunden wird. Dieses führt zur Ausgabe von drei Punkten. Beispielsweise:

```
booting my program ...
```

Jetzt gewinnen die zuerst gelesenen vier Byte ihre Bedeutung. Ist der Zähler $af ungleich Null, so werden weitere Sektoren von Diskette gelesen und ab der in $ac/ $ad definierten Startadresse abgespeichert. Ist zum Beispiel der Zähler gleich 2, so werden noch die Sektoren 2 und 3 der Spur 0 nachgeladen.
Sind alle Blöcke geladen, beziehungsweise war der Zähler für die nachzuladenden Blöcke sowieso Null, geht es im Kassettenpuffer hinter dem zuletzt gelesenen Nullbyte weiter. An dieser Stelle kann ein File-Name im ursprünglichen BOOT-Sektor folgen, der wieder mit einem Nullbyte abgeschlossen ist. Ist die Länge des File-Namens ungleich Null, wird das entsprechende File durch die Kernal-Routine LOAD in den Speicher geladen. Nachdem dies geschehen ist, beziehungsweise wenn kein File-Name angegeben war, geht es hinter dem Nullbyte des File-Namens im Kassettenpuffer weiter. Die BOOT-Routine des Kernals übergibt die Kontrolle an ein Anwenderprogramm, das an dieser Stelle im

BOOT-Sektor beginnt. Dieses Programm wird in der Regel ein geladenes Programm starten.

3.4.4 Fehlererkennung und Fehlerbehandlung

Wie schon bemerkt, sind alle bisherigen Beispiele wegen der besseren Übersichtlichkeit ohne Berücksichtigung eventuell auftretender Fehler geschrieben. Dies soll aber keinesfalls bedeuten, daß man die Fehlerabfrage einfach links liegen lassen sollte, ganz im Gegenteil. Ein Programm, das beim geringsten auftretenden Fehler sofort seinen Geist aufgibt, ist doch wohl höchst ärgerlich. Dies ist für mich der Anlaß, der Fehlererkennung und Fehlerbehandlung einen eigenen Abschnitt zu widmen.
Wir müssen zunächst auch bei der Fehlererkennung unterscheiden, ob wir direkt mit LISTEN, SECLST etc. programmieren oder mit logischen File-Nummern.

3.4.4.1 Fehlererkennung bei direkter Programmierung des IEC-Bus

Der erste Fall ist auch in der Fehlererkennung umständlicher. Die Routinen, die hier eingesetzt werden, liefern keine Rückmeldung, ob während ihrer Benutzung ein Fehler aufgetreten ist oder nicht.
Die einzige Möglichkeit, einen eventuellen Fehler erkennen zu können, besteht in der Abfrage des STATUS-Bytes, wobei man berücksichtigen muß, daß STATUS nicht automatisch gelöscht wird.
Der allererste Schritt, bevor man eine Datenübertragung mit LISTEN, TALK usw. einleitet, muß daher korrekterweise immer sein, den STATUS zu löschen:

```
lda #0
sta status
```

Nach Aufruf der Routinen LISTEN und TALK, oder aber nach Aufruf von SECLST und SECTLK wird dann das STATUS-Byte abgefragt. Es ist hier an dieser Stelle nur das Bit 7 (Device not present) interessant, andere Fehler können noch nicht erwartet werden. Die Fehlererkennung nach SECLST und SECTLK kann also so aussehen:

```
bit status
bmi fehler
```

Fehler während der Übertragung mit IECIN und IECOUT werden auf dieselbe Art entdeckt. Hier können allerdings auch die anderen Fehler (siehe Beschreibung des STATUS-Bytes) auftreten. Insbesondere ist beim Datenempfang auf das EOF-Bit innerhalb des Status zu achten. Dieses Bit kann man leicht erkennen durch

```
bit status
bvs eof
```

Wer mit Gürtel und Hosenträger arbeiten will, kann auch nach UNTLK und UNLST noch einmal den Status abfragen, in der Regel wird dies aber unnötig sein.

Allen Fehlern ist gemeinsam, daß ihr Auftreten die Datenübertragung von seiten des Kernals beendet. Man erhält dann beim Weiterlesen vom Bus zum Beispiel unsinnige Ergebnisse. Schon aus diesem Grund empfiehlt es sich unbedingt, gerade die Fehlererkennung gewissenhaft zu betreiben.

3.4.4.2 Fehlererkennung beim Arbeiten mit logischen File-Nummern oder LOAD/SAVE

Wenn wir die Routinen OPEN, CHKIN usw. benutzen, haben wir es bei der Fehlererkennung erheblich leichter.

Die Routinen

OPEN
CHKIN
CKOUT
LOAD
SAVE

zeigen nämlich im Carry-Flag bei der Rückkehr zum aufrufenden Programm an, ob ein Fehler aufgetreten ist. Dies wird durch eine STATUS-Abfrage innerhalb dieser Routinen möglich. Das gesetzte Carry-Flag weist immer auf einen Fehler hin.
Auch eine folgende Statusabfrage, um zu erkennen, um welchen Fehler es sich gehandelt hat, kann man sich sparen. Bei der Rückkehr mit gesetztem Carry befindet sich immer eine Fehlernummer im Akkumulator, anhand derer im aufrufenden Programm weitere Verzweigungen erfolgen können. Es kommt sogar noch besser: Mit Hilfe eines Flags kann man das Betriebssystem veranlassen, gleich selbst eine entsprechende Fehlermeldung der Art

```
I/O Error #...
```

auf den Bildschirm auszugeben. Das entsprechende Flag liegt in der Zeropage:

MSGFLG = $9d

Ein gesetztes Bit 6 des Flags erlaubt dem Kernal die Ausgabe der Fehlermeldung. Die Ausgaberoutine selbst liegt übrigens bei $f699, für diejenigen, die einmal mit dem Disassembler hineinsehen möchten.

Das Kernal kennt Fehlernummern von 0 bis 9. Die Bedeutung der einzelnen Fehlernummern können Sie der folgenden Tabelle entnehmen:

Fehlernummer	entsprechende Meldung, die bei BASIC ausgegeben würde
0	break (durch Stop-Taste)
1	too many Files
2	file open
3	file not open
4	file not found
5	device not present
6	not input file
7	not output file
8	missing filename
9	illegal device number

Ist ein Fehler wie oben gefunden worden, wird die Datenübertragung mit CLRCH beendet. Das CLOSE muß man gegebenenfalls selbst erledigen.
Fehler bei BASIN, BSOUT und den anderen Routinen können wieder nur über eine Abfrage des Statusbytes erkannt werden, mithin auch das Auftreten von EOF.
Auch die bekannten Meldungen »loading«, »searching«, »press play on tape« usw. werden vom Betriebssystem wahlweise erzeugt. Um eine Ausgabe dieser Meldungen zu gestatten, wird das Bit 7 von MSGFLG gesetzt. Die Ausgaberoutine befindet sich an der Adresse $f71e.

3.5 Kernal-Routinen der Sprungleiste $ff47 - $fff3

Die Kernal-Routinen, die in der oberen Sprungleiste des Kernal-ROMs zusammengefaßt sind, decken fast jeden Bedarf ab, der in eigenen Maschinenprogrammen an Betriebssystemfunktionen entstehen dürfte. Die Sprungleiste enthält im wesentlichen alle zur I/O notwendigen Unterprogramme. Die Lage der einzelnen Funktionen innerhalb der Sprungleiste wird sich auch bei eventuellen ROM-Änderungen nicht verändern, deshalb sollte man, wenn irgend möglich, nur die Einsprungpunkte in der Sprungleiste benutzen.

Eine Erläuterung zur Notation der Routinen noch: IN soll die Aufrufparameter bezeichnen, die Parameter, die das jeweilige Unterprogramm bei seinem Aufruf benötigt. OUT bezeichnet entsprechend die Parameter, die zurückgeliefert werden. COM zeigt eine Bemerkung an, zum Beispiel, welche Register von der Routine nicht verändert werden oder ähnliches.

FSTMOD = $ff47

Vorbereiten der CIAs 1 und 2 für langsame beziehungsweise schnelle serielle Übertragung
IN: sec, clc
Bei gesetztem Carry-Flag wird das Kontrollbit für den schnellen seriellen Modus im MCR der MMU ($d505, Bit 3) gesetzt. Alle Interrupts durch den CIA 1 werden verhindert. Timer A des CIA 1 wird mit dem Startwert 4 geladen und gestartet. Betriebsart des Timers ist Toggle (Bit 2 CRA = 1) und Continuous, der serielle Port SP ist als Ausgang eingestellt. Bei gelöschtem Carry wird die normale Einstellung des Timers A des CIA 1 wiederhergestellt.
Sendet man über SP im ersten Fall ein DRF-Signal (Device Requests Fast = $ff), indem man SP mit $ff lädt und über Timer A das Byte hinaustaktet, kann man an der Antwort der Floppy erkennen, ob es eine schnelle oder langsame ist. Folgt nach Wiederherstellung des alten CIA-Zustandes ein HRF (= $00) über das Bit 7 des PRA des CIA 2, so handelt es sich um eine schnelle Floppy.
Das genaue Vorgehen bitte ich den LISTEN- beziehungsweise TALK-Routinen des Kernals bei Interesse zu entnehmen.
OUT: –
COM: X- und Y-Register werden nicht verändert

GACLOSE = $ff4a

Schließen aller Files eines Gerätes
IN: Akku = Gerätenummer
Es werden alle Files geschlossen und aus den Tabellen für logische File-Nummern etc. gestrichen, die die im Akkumulator übergebene Gerätenummer enthalten. Aktuelle Ein- und Ausgabegeräte werden gegebenenfalls auf Default gesetzt (Tastatur und Bildschirm).
OUT: –
COM: Y-Register unbeeinflußt

C64MODE = $ff4d

Umschaltung wie bei GO 64
IN: –
Es wird vom 128er- in den 64er-Modus übergegangen. Bisher hat noch niemand einen Rückweg vom 64er- in den 128er-Modus gefunden, aber wer weiß? Das 64er-Betriebssystem wird am RESET-Vektor angesprungen.
OUT: –

DMACALL = $ff50

Speicherzugriff durch externe Geräte

IN: X, Y

Unter DMA hat man einen direkten Zugriff externer Geräte auf den Computer-Speicher zu verstehen. Diese Routine würde bei einer RAM-Floppy, die irgendwann einmal für den C128 erscheinen soll, verwendet werden. In diesem Fall würde im I/O-Bereich bei $df01 ein Steuerregister der RAM-Disk erwartet, das über eine Routine in der erweiterten Zeropage ($3f0) angesprochen wird. In der normalen Ausstattung des 128ers liegt bei $df00 nichts, die Routine DMACALL hat also keine Bedeutung.

OUT: –

BOOTCALL = $ff53

Booten einer Diskette

IN: Akku = Laufwerknummer in ASCII, X = Gerätenummer

Durch diese Routine wird ein BOOT-Load versucht, wie es bei einem Reset oder beim Einschalten des Computers erfolgt. Hierzu wird der Sektor 0 der ersten Spur der durch Akku und X-Register bestimmten Diskette in den Kassettenpuffer ab $b00 geladen. Sind die ersten drei Zeichen des gelesenen Blocks gleich CBM, handelt es sich um einen BOOT-Sektor. Eine nähere Beschreibung des BOOTens finden Sie unter den Kapiteln Laden und Abspeichern.

OUT: –

PHOENIX = $ff56

System-Kaltstart

IN: –

Kaltstart des 128ers. Bei eingesteckten Modulen werden diese initialisiert, ansonsten wird nach BOOTCALL verzweigt, das versucht, eine Diskette von Laufwerk 0, Gerät 8, zu booten.

OUT: –

LKUPLA = $ff59

Suchen einer logischen File-Nummer

IN: A = logische File-Nummer

LKUPLA sucht eine logische File-Nummer in der entsprechenden Tabelle. Wird ein Eintrag mit der logischen File-Nummer gefunden, so werden die Zeropage-Adressen LA, SA und FA mit den Einträgen aus den Tabellen für Geräteadressen usw. besetzt. Akkumulator sowie X- und Y-Register sind mit logischer File-Nummer, Gerätenummer und Sekundäradresse geladen. Das Carry dient als Fehler-Flag. Ist es gesetzt, wurde kein Eintrag entdeckt.

OUT: Carry, A, X, Y

COM: STATUS wird durch LKUPLA gelöscht.

LKUPSA = $ff5c

Suchen einer Sekundäradresse

IN: Y = Sekundäradresse

LKUPSA sucht entsprechend zu LKUPLA einen Eintrag mit der Sekundäradresse, die im Y-Register übergeben wird. Ob ein Eintrag gefunden wurde, wird wieder im Carry-Flag angezeigt. Die Register enthalten alle Tabelleneinträge, wenn die Routine fündig geworden ist.

OUT: Carry, A, X, Y

COM: STATUS wird nicht gelöscht.

SWITCH = $ff5f

Umschaltung 40/80-Zeichen

IN: –

Umschaltung zwischen dem VIC- und dem VDC-Bildschirm. Der passive und der aktive Bildschirm-Speicher werden ausgetauscht. Das MODE-Flag wird entsprechend umgedreht.

OUT: –

DLCHR = $ff62

Zeichensatz in den VDC-Speicher bringen

IN: –

Kopieren des Zeichensatz-ROMs in das VDC-RAM.

OUT: –

PFKEY = $ff65

Programmieren einer Funktionstaste

IN: X = Nummer der Funktionstaste
Y = Länge des Tasten-Strings
A = Lowbyte des Pointers, in dem die Adresse des Tasten-Strings steht

Die Routine PFKEY erlaubt die Neudefinition der Funktionstastenbelegung. Außer den drei Registern werden drei aufeinanderfolgende Bytes in der Zeropage benötigt, um die notwendigen Parameter an PFKEY zu übergeben. Der Zeiger auf diese drei Byte wird im Akku übermittelt. Die ersten beiden Byte des Pointers enthalten die Adresse, an der der neue Tasten-String zu finden ist, in der Reihenfolge Low-/Highbyte. Das dritte Byte enthält den Konfigurationsindex dazu.

Beispiel einer Funktionstastenbelegung:

```
          .base $1300

 .define pointer = $fc
 .define pfkey = $ff65
```

```
            lda #<(string) ; Adresse des neuen Strings
            ldx #>(string)
            sta pointer    ; in die ersten beiden Bytes
            stx pointer+1  ; des Pointers
            lda #15        ; Index 15 = Bank 0, alle ROMs
            sta pointer+2  ; in das dritte Byte
            lda #pointer   ; Zeiger auf Pointer
            ldx #1         ; Nummer der Funktionstaste
            ldy #4         ; Länge des neuen Strings
            jmp pfkey      ; Definition und rts

string      .byte "test"   ; neuer Tasten-String
```

Näheres zur Funktionstastenbehandlung durch das Betriebssystem können Sie im Kapitel 3.1 nachlesen.
OUT: –
COM: Rückkehr immer mit clc

SETBNK = $ff68

Setzen der Bank-Nummer für File-Namen und Load/Save
IN: A = RAM-Bank-Nummer für Load, Save, Verify
X = RAM-Bank-Nummer des File-Namens
SETBNK muß vor jedem OPEN, LOAD oder SAVE aufgerufen werden, damit sichergestellt ist, daß die Zeropage-Adressen

BA = $c6 Bank für LSV-Aufrufe
FNBNK = $c7 Bank des File-Namens

richtig gesetzt sind.
OUT: –
COM: Y bleibt unverändert

GETCFG = $ff6b

Holen der zu einem Konfigurationsindex gehörenden Konfiguration
IN: X = Konfigurationsindex
Mit GETCFG wird die Konfiguration aus der Tabelle $f7f0 gelesen, die dem im X-Register übergebenen Index entspricht. Die Tabelle ist zwar nur 16 Byte lang (Indizes von 0 bis 15), aber vielleicht findet man in den nachfolgenden Bytes noch eine Konfiguration, die man benutzen kann. Zu den Konfigurationsindizes mehr in Kapitel 2.1.1.
OUT: A = Konfiguration
COM: Y wird nicht verändert.

Die folgenden fünf Routinen sind schon ausführlich in Kapitel 2 behandelt worden. JSRFAR und JMPFAR in der Sprungleiste stellen einfache Sprünge zu den eigentlichen Routinen in der Common Area dar.

JSRFAR = $ff6e
JMPFAR = $ff71
INDFET = $ff74
INDSTA = $ff77
INDCMP = $ff7a

PRIMM = $ff7d

Ausgabe einer Zeichenkette mit BSOUT
IN: –
PRIMM steht für Print immediately. Die Routine gibt eine Zeichenkette aus, die direkt hinter dem Befehl »jsr primm« beginnt und mit einem Nullbyte abgeschlossen ist.

```
jsr primm

.byte "text",0
```

Nach der Ausgabe wird das Maschinenprogramm hinter dem Nullbyte weitergeführt. Da diese Routine so schön bequem zur Ausgabe von Strings ist, möchte ich noch das Listing von PRIMM zeigen, in dessen Kommentierung die Arbeitsweise der Routine ersichtlich wird:

```
$fa17   pha          ; Akku auf den Stack retten
        txa          ; X-Register
        pha          ; retten
        tya          ; Y-Register
        pha          ; retten
        ldy #0       ; Offset für indirektes Laden
$fa1e   tsx          ; Stackpointer nach X
        inc $104,x   ; Lowbyte der RTS-Adresse + 1
        bne $fa27    ; nicht Null: kein Überlauf
        inc $105,x   ; Highbyte der RTS-Adresse + 1
$fa27   lda $104,x   ; anschließend die erhöhte RTS-
        sta $ce      ; Adresse in den Zeropage-Pointer
        lda $105,x   ; $ce/$cf, der Pointer weist
        sta $cf      ; hinter das JSR PRIMM
        lda ($ce),y  ; Zeichen holen
        beq $fa3a    ; Null: Ende-Kennzeichen
        jsr $ffd2    ; BSOUT: allgemeine Zeichenausg.
        bcc $fa1e    ; RTS von BSOUT immer clc
```

```
$fa3a   pla         ; Nullbyte gefunden:
        tay         ; Register wiederherstellen
        pla
        tax
        pla
        rts         ; Ende und rts
```

Wie man sieht, beeinflußt PRIMM keines der Register, da diese während der Zeichenausgabe auf den Stack gelegt werden. Damit die Zeichenausgabe aber wie gewünscht funktioniert, muß man auf einen Sachverhalt immer achtgeben: Der auszugebende String muß von PRIMM aus »sichtbar« sein. Liegt der String beispielsweise unter einem der Basic-ROMs, muß vor dem PRIMM-Aufruf das ROM ausgeschaltet werden. Strings unterhalb des Kernal-ROMs können von PRIMM auf keinen Fall ausgegeben werden.
OUT: –
COM: Alle Register werden gerettet, EQUAL-Flag beim rts immer gesetzt.

CINT = $ff81
Kaltstart des Kernal-Editors
IN: –
Kernal-Editor-Kaltstart. Beide Videocontroller werden initialisiert, alle Flags auf Standard zurückgesetzt. Der eingeschaltete Bildschirm ergibt sich aus dem Zustand der 40/80-Zeichen-Taste.
OUT: –

IOINIT = $ff84
Kaltstart aller I/O-Geräte
IN: –
RESET aller angeschlossenen Geräte. Die Register der CIAs, Videocontroller und des SID werden mit den Standardwerten belegt. Über die RESET-Leitung des IEC-Bus werden Floppy und Drucker in den Einschaltzustand versetzt.
OUT: –

RAMTAS = $ff87
Löschen der Zeropage, Setzen der Systemvektoren
IN: –
Löschen der Zeropage mit Ausnahme der Prozessorport-Register (Zeropage 0 und 1). Ober- und Untergrenze des Speichers werden durch die beiden später beschriebenen Routinen MEMTOP und MEMBOT auf $ff00 und $1c00 gelegt. Initialisierung der Zeiger auf Kassettenpuffer ($b2/$b3 = $b00) und der Zeiger auf den RS232-Ein- beziehungsweise Ausgabepuffer ($c8/$c9 = $c00; $ca/$cb = $d00). Außerdem wird der RESTART-Vektor $a00/$a01 auf den BASIC-Einsprung $4000 gebracht.
OUT: –

RESTOR = $ff8a

Initialisierung der Kernal-Vektoren

IN: –

Kopieren der Kernal-Vektoren, die im Bereich von $314 bis $332 zu finden sind, aus dem ROM ab $e073 in die Common Area. Dies kann recht nützlich sein, wenn einige Vektoren durch eigene Programme verbogen sind.

OUT: –

VEKTOR = $ff8d

Kopieren der Systemvektoren

IN: Carry-Flag
X = Lowbyte der Kopieradresse
Y = Highbyte der Adresse

Je nach Zustand des Carry-Flags kopiert diese Routine die bei RESTOR angesprochenen Kernal-Vektoren entweder von der in X und Y angegebenen Adresse in die Common Area (Carry gelöscht) oder andersherum (Carry gesetzt).

OUT: –

SETMSG = $ff90

Flag für Systemmeldungen setzen

IN: A = Wert für MSGFLG = $9d

SETMSG besteht nur darin, den im Akku übergebenen Wert in MSGFLG abzuspeichern. Gesetztes Bit 6 in MSGFLG erlaubt dem Kernal die Ausgabe von Fehlermeldungen, Bit 7 erlaubt die Ausgabe von Meldungen wie »loading«, »searching« etc.

OUT: –

SECLST = $ff93

Sekundäradresse nach LISTEN senden

IN: A = Sekundäradresse

SECLST sendet die Sekundäradresse nach LISTEN. Es erfolgt keine Fehlerrückmeldung im Prozessorstatus. Die Sekundäradresse wird temporär in

BSOUR = $95

aufbewahrt.

OUT: –

COM: Beim RTS ist das Carry immer gelöscht.

SECTLK = $ff96

Sekundäradresse nach TALK senden

IN: A = Sekundäradresse

Es gilt dasselbe wie für SECLST. SECTLK arbeitet nur dann, wenn das STATUS-Byte kein EOF ausweist.
OUT: –
COM: Carry ist immer clear bei der Rückkehr.

MEMTOP = $ff99

Setzen der Obergrenze des Systemspeichers
IN: clc, sec
X = Lowbyte der Obergrenze
Y = Highbyte
Durch MEMTOP wird die Obergrenze des Systemspeichers festgelegt – Aufruf mit gelöschtem Carry – oder gelesen – Aufruf mit gesetztem Carry. Die Grenze zeigt für die sogenannte System-Bank, das ist die RAM-Bank, in dem das BASIC sozusagen zu Hause ist, die Bank 0, hinter die oberste Adresse, die das System (BASIC) noch benutzen darf. Der Normalwert für die Obergrenze ist $ff00, das heißt der BASIC-Programmtext darf bis $feff unter das Kernal-ROM reichen. In der erweiterten Zeropage findet sich dieser Wert im Zeiger

MEMSIZ = $a07/ $a08

wieder. Durch Herabsetzen der Obergrenze des BASIC-Speichers gewinnt man Speicherplatz für eigene Programme, der von BASIC nicht beeinträchtigt werden kann.
OUT: –
COM: Carry wird nicht verändert, Akku ist unbenutzt.

MEMBOT = $ff9c

Setzen der Untergrenze des Systemspeichers
IN: sec, clc
X = Lowbyte der Untergrenze
Y = Highbyte der Untergrenze
Analog zu MEMTOP wird in MEMBOT die Untergrenze des Systemspeichers definiert. Im Normalfall beträgt die Untergrenze $1c00. Durch entsprechendes Heraufsetzen der Grenze gewinnt man wieder Raum für eigene Programme, wobei man allerdings darauf achten muß, daß ein eingeschalteter Grafikschirm das RAM bis $4000 beansprucht. In der erweiterten Zeropage befindet sich die Untergrenze des Systemspeichers in

MEMSTR = $a05/ $a06

OUT: –

KEY = $ff9f

Tastaturmatrix-Abfrage
IN: –

Die Tastaturmatrix-Abfrage wurde eingehend in Kapitel 3.1 behandelt. Ihre Wirkungsweise kann dort nachgelesen werden.
OUT: Tastaturpuffer und Zeiger in der Zeropage wie in 3.1 beschrieben.

SETTMO = $ffa2

Setzen des Timeout-Flags
IN: Akkumulator
Das Timeout-Flag

TIMOUT = $a0e

wird bei normaler Ausstattung des C128 nicht benutzt. Es gewinnt erst an Bedeutung, wenn eine IEEE-Erweiterung (paralleler Bus) angeschlossen wird. Die Routine kann man also im Normalfall vergessen.
OUT: –

Zu den meisten der folgenden Routinen siehe auch Kapitel 3.4!

IECIN = $ffa5

Eingabe vom IEC-Bus
IN: –
Nachdem mit TALK und SECTLK ein am IEC-Bus angeschlossenes Gerät zum Senden von Daten veranlaßt wurde, kann mit IECIN ein Byte vom Bus abgeholt werden. IECIN ist eine Unterroutine der allgemeineren Eingaberoutine BASIN.
OUT: A = Byte vom IEC-Bus
COM: X und Y werden gerettet beziehungsweise bleiben unverändert. Das Carry ist beim rts immer gelöscht.

IECOUT = $ffa8

Ausgabe auf IEC-Bus
IN: = auszugebendes Byte
Entsprechend zu IECIN dient IECOUT der Ausgabe eines Bytes auf den Bus, nachdem mit LISTEN und SECLST ein Gerät zum Datenempfang vorbereitet wurde. IECOUT ist Unterroutine von BSOUT für den Fall, daß das aktuelle Ausgabegerät ein Gerät am IEC-Bus ist (Gerätenummer größer als 3).
OUT: –
COM: Alle Register, auch der Akkumulator, bleiben erhalten beziehungsweise werden über den Stack gerettet. Das Carry ist beim rts immer clear.

UNTALK = $ffab

Abschluß des Datenempfangs
IN: –

Am Ende einer mit TALK und SECTLK eingeleiteten Datenübertragung von einem Peripheriegerät zum Computer muß der UNTALK-Befehl über die gleichnamige Routine gesendet werden. Der UNTALK-Befehl gibt den IEC-Bus wieder frei, so daß gegebenenfalls andere Geräte angesprochen werden können. UNTALK löscht das Flag für die schnelle serielle Übertragung.
OUT: –
COM: X- und Y-Register werden gerettet.

UNLIST = $ffae
Abschluß einer Datensendung
IN: –
Wie der Datenempfang durch UNTALK abgeschlossen wird, so wird eine Datensendung des Computers an ein Peripheriegerät durch das Senden von UNLISTEN in dieser Routine beendet. Auch durch diese Routine wird das Flag für die schnelle serielle Übertragung gelöscht.
OUT: –
COM: X- und Y-Register sind nach Aufruf der Routine unverändert.

LISTEN = $ffb1
Beginn einer Datensendung
IN: A = Gerätenummer des angesprochenen Peripheriegerätes
Durch die LISTEN-Routine wird eine Datensendung des Computers an ein Gerät am IEC-Bus eingeleitet. LISTEN sendet dazu die im Akku übergebene Gerätenummer, die zusammen mit einem Zusatz im oberen Halbbyte zu einer sogenannten Primäradresse zusammengesetzt wird, die gleichzeitig die Absicht des Computers anzeigt, eine Datensendung zu beginnen. Durch LISTEN wird selbständig eine Abfrage auf die Möglichkeit einer schnellen seriellen Übertragung durchgeführt und das Ergebnis im Flag für diese Übertragungsart temporär abgelegt.
OUT: –
COM X- und Y-Register bleiben unverändert. Das Carry ist bei der Rückkehr immer clear.

TALK = $ffb4
Beginn eines Datenempfangs
IN: A = Gerätenummer des Geräts, das Daten senden soll.
Genau wie beim Senden von Daten setzt TALK aus der Geräteadresse und einem Zusatz eine Primäradresse zusammen, die dem angesprochenen Peripheriegerät anzeigt, daß es Daten senden soll. In der Regel folgt der Primäradresse wie bei LISTEN noch eine Sekundäradresse, die unterschiedliche Arbeitsmodi des Geräts anspricht. Auch von der TALK-Routine wird die Möglichkeit einer schnellen seriellen Übertragung selbständig abgefragt.
OUT: –
COM: X- und Y-Register bleiben unbeeinflußt, das Carry ist beim rts clear.

READST = $ffb7

Lesen des System-Status

IN: –

Je nachdem, welche Geräteadresse sich aktuell in der Zeropage-Zelle GA = $ba befindet, wird entweder der RS232-Status (RSSTAT = $a14) oder das STATUS-Byte für IEC- und Kassettenbetrieb (STATUS = $90) ausgelesen. Der RS232-Status wird durch das Lesen außerdem gelöscht.

OUT: A = Status

COM: X und Y unverändert.

SETPAR = $ffba

Setzen von Geräteadresse, logischer File-Nummer und Sekundäradresse.

IN: A = logische File-Nummer
X = Gerätenummer
Y = Sekundäradresse ohne Zusatz (bei der Floppy = Kanalnummer)

Diese Routine ist dazu da, die oben genannten Parameter in die passenden Zeropage-Zellen zu bringen, mehr tut sie nicht. Diese Zeropage-Zellen sind:

LA = $b8	logische File-Nummer
SA = $b9	aktuelle Sekundäradresse
FA = $ba	aktuelle Gerätenummer

Vor jedem Aufruf der OPEN-Routine müssen diese Parameter, soweit erforderlich, gesetzt sein.

OUT: –

SETNAM = $ffbd

Setzen der Parameter des File-Namens

IN: A = Länge des File-Namens
X = Lowbyte der Adresse des File-Namens
Y = Highbyte der Adresse

Ebenso wie die Gerätenummer und Sekundäradresse für ein OPEN vorher gesetzt werden müssen, so müssen auch die Parameter des File-Namens, soweit vorhanden, in der Zeropage abgelegt werden. Dies erfolgt durch SETNAM. In der Zeropage finden sich die Parameter an folgenden Adressen wieder:

FNLEN = $b7	Länge des File-Namens
FNADR = $bb/$bc	Adresse des Namens
FNBNK = $c7	RAM-Bank des File-Namens

Ist kein File-Name vorgesehen, muß FNLEN auf Null gesetzt werden, damit die OPEN-Routine dies berücksichtigt. Die RAM-Bank, in der sich der File-Name befindet, wird durch SETBNK gesetzt.

OUT: –

OPEN = $ffc0

Öffnen und Tabelleneintrag eines Files

IN: Parameter werden durch SETBNK, SETPAR und SETNAM in der Zeropage abgelegt

Durch die OPEN-Routine wird das mit SETPAR etc. beschriebene File geöffnet und unter der angegebenen logischen File-Nummer in eine Tabelle eingetragen, so daß bei weiterer Bezugnahme auf das File nur die Angabe der logischen File-Nummer notwendig ist. Durch OPEN wird noch kein Übertragungskanal freigeschaltet, dies muß erst durch CHKIN oder CKOUT geschehen, bevor man Daten empfangen oder senden kann.

OUT: -

COM: Gesetztes Carry zeigt einen Fehler an (s. Kapitel 3.4.4).

CLOSE = $ffc3

Schließen einer Datei und Entfernung ihres Tabelleneintrags

IN: clc, sec
A = logische File-Nummer

Eine durch OPEN geöffnete Datei wird durch die CLOSE-Routine wieder geschlossen. Ihr Eintrag in die Tabellen der logischen File-Nummern usw. wird entfernt. Das Carry-Flag spielt eine Rolle, wenn der Kommandokanal zu einer Floppy (Kanalnummer 15) geöffnet und jetzt durch CLOSE das entsprechende File geschlossen werden soll. Ist das Carry gesetzt, bewirkt dies, daß das Senden der Close-Sekundäradresse über LISTEN, SECLST, UNLIST vermieden wird.

OUT: -

COM: Das Carry-Flag ist beim rts immer gesetzt. Es erfolgt keine Fehleranzeige.

CHKIN = $ffc6

Eingabekanal schalten

IN: X = logische File-Nummer

Nach erfolgtem OPEN eines Files kann mit Hilfe der CHKIN-Routine das angesprochene Peripheriegerät zum aktuellen Eingabegerät gemacht werden, so daß die Routine BASIN Daten von diesem Gerät holen kann.

OUT: -

COM: Gesetztes Carry weist wie bei OPEN auf einen Fehler hin (s. Kapitel 3.4.4). Das Y-Register wird in der Routine nicht benutzt.

CKOUT = $ffc9

Ausgabegerät schalten

IN: X = logische File-Nummer

Genau wie die Eingabe auf verschiedene Geräte gelegt werden kann, kann dies auch mit der Ausgabe geschehen. CKOUT legt das aktuelle Ausgabegerät fest, so daß mit BSOUT in ein File geschrieben werden kann. Folgendes ist zu beachten: Es ist zwecklos zu ver-

suchen, gleichzeitig einen Eingabe- und einen Ausgabekanal zu einem Gerät am IEC-Bus schalten zu wollen. Der IEC-Bus kann nur eine der Aufgaben auf einmal erfüllen.
OUT: –
COM: Gesetztes Carry bedeutet wieder einen Fehler, das Y-Register wird nicht verändert.

CLRCH = $ffcc

Rücksetzen des Ein- und Ausgabegerätes auf Default
IN: –
Die mit CHKIN und CKOUT gewählten Ein- beziehungsweise Ausgabegeräte werden durch CLRCH wieder rückgängig gemacht. Standard-Eingabegerät ist wieder die Tastatur, Standard-Ausgabegerät ist der Bildschirm. CLRCH sendet im Fall, daß der IEC-Bus angesprochen war, ein UNLIST oder UNTALK. Die CLRCH-Routine kommt immer vor der CLOSE-Routine, mit der ein File endgültig geschlossen wird. Folgt kein CLOSE, kann die Ein- oder Ausgabe jederzeit wieder mit CHKIN oder CKOUT auf das File gelegt werden.
OUT: –
COM: Keine Fehleranzeige im Carry. Der Akku ist beim rts immer gleich Null, das EQUAL-Flag ist gesetzt.

BASIN = $ffcf

Allgemeine Routine >Zeichen holen<
IN: –
BASIN holt in Abhängigkeit vom eingestellten aktuellen Eingabegerät ein Zeichen von einem Peripheriegerät. Dieses Gerät kann auch der Bildschirm sein. Ist die Tastatur als aktuelles Eingabegerät ausgeschaltet, erfüllt BASIN eine spezielle Funktion, die ausführlich in Kapitel 3.2.5.2 (Eingaben mit blinkendem Cursor) beschrieben ist. Ist ein Gerät am IEC-Bus als Eingabegerät geschaltet, so verzweigt BASIN in die Routine IECIN und es gilt das dort Gesagte.
OUT: A = gelesenes Zeichen
COM: X- und Y-Register werden gerettet. Das Carry ist beim rts immer gelöscht.

BSOUT = $ffd2

Allgemeine Ausgabe eines Zeichens an ein Peripheriegerät
IN: A = auszugebendes Zeichen
Je nach dem in DFLTO = $9a eingestellten aktuellen Ausgabegerät verzweigt die BSOUT-Routine in verschiedene Unterroutinen, die die Ausgabe auf verschiedene Geräte übernehmen, so zum Beispiel PRINT für die Bildschirmausgabe oder IECOUT für die Ausgabe auf den IEC-Bus. Die Tastatur kann selbstverständlich nicht als Ausgabegerät gewählt werden. Wird nicht durch CKOUT ausdrücklich etwas anderes vereinbart, so ist der Bildschirm das Ausgabegerät des Computers.
OUT: –

COM: Alle Register, auch der Akkumulator, bleiben erhalten. Bei der Rückkehr von BSOUT ist das Carry-Flag immer gelöscht.

LOAD = $ffd5

Laden von Programmen

IN: A = Load/ Verify-Schalter
X = Ladeadresse Lowbyte (optional)
Y = Ladeadresse Highbyte

Durch LOAD wird ein Programm von Diskette oder Kassette in den Speicher geladen beziehungsweise verifiziert. Über den Arbeitsmodus entscheidet der Inhalt des Akkumulators beim Aufruf von LOAD. Ist er gleich 1, wird verifiziert, ist er 0, wird geladen. LOAD fragt wie TALK und LISTEN selbständig die Möglichkeit einer schnellen Übertragung ab. Die weiteren für LOAD benötigten Parameter werden durch die Routinen SETBNK, SETPAR und SETNAM gesetzt. Eine gesetzte Sekundäradresse von Null veranlaßt ein relatives Laden, das heißt das Programm wird an die Adresse geladen, die in X- und Y-Register übergeben werden kann. Ist die Sekundäradresse ungleich Null, wird absolut geladen, das heißt an die Adresse, die im File selbst vermerkt ist. Die RAM-Bank, in die geladen wird, wird durch SETBNK in die Speicherzelle

BA = $c6 Bank für Load/ Save und Verify

geschrieben. Zu LOAD und SAVE siehe auch Kapitel 3.4.3.

OUT: X = Lowbyte der Endadresse des Ladens (Adresse hinter letztem abgespeicherten Byte).
Y = Highbyte der Adresse

COM: Eventuelle Fehler werden wie bei OPEN durch ein gesetztes Carry angezeigt.

SAVE = $ffd8

Abspeichern eines Programms

IN: A = Lowbyte eines Pointers in der Zeropage
X = Lowbyte der Endadresse des Programms
Y = Highbyte der Endadresse

Auch bei SAVE werden die Parameter wie Gerätenummer oder der File-Name durch die Routinen SETBNK, SETPAR und SETNAM gesetzt. Der schnelle serielle Modus wird wie immer selbständig überprüft. Die Startadresse des abzuspeichernden Bereichs wird in einer frei wählbaren Zeropage-Adresse übergeben, der Akkumulator enthält einen Zeiger auf diese Adresse. Die Endadresse des Bereiches wird in X und Y übergeben. Die RAM-Bank, in der der Bereich liegt, wird durch SETBNK gesetzt. Näheres zu SAVE im Kapitel 3.4.3.2.

OUT: –

COM: Das Carry weist wieder auf Fehler hin.

SETTIM = $ffdb

Setzen der Systemuhr

IN: A = Lowbyte der Systemuhr
X = Middlebyte der Uhr
Y = Highbyte

Die Systemuhr TI wird in den drei Zeropage-Byte

TIME = $a0 / $a1 / $a2

geführt. Sie wird etwa jede 1/60 Sekunde während des Interrupts um 1 weitergesetzt. Ist ein Wert von 86400 * 60 = 1 Tag erreicht, wird die Uhr wieder auf Null zurückgesetzt. Da das Weiterzählen der Uhr während des Interrupts geschieht, der zum Beispiel durch Benutzung von BASIC-Befehlen zur Sprite-Behandlung verzögert wird, ist die Systemuhr wesentlich ungenauer als die in die CIAs eingebauten Echtzeituhren.

OUT: –

RDTIM = $ffde

Lesen der Systemuhr

IN: –

Diese Routine liest die Zeropage-Bytes von TIME in die Prozessorregister.

OUT: A = Lowbyte von TIME
X = Middlebyte
Y = Highbyte der Systemuhr

STOP = $ffe1

Abfrage der STOP-Taste

IN: –

Die Abfrage auf die gedrückte STOP-Taste geschieht durch Kontrolle des Zeropage-Flags

STKEY = $91

Dieses Flag wird während des NMI-Interrupts in einer eigenen Tastaturmatrix-Abfrage (Adresse im ROM = $f63d) auf $7f gesetzt, wenn die STOP-Taste gedrückt war. Übrigens werden noch weitere Tasten von dieser Abfrage erfaßt. STKEY enthält zum Beispiel $df, wenn die CBM-Taste gedrückt und $fb, wenn die Control-Taste betätigt wurde. Die STOP-Routine führt ein CLRCH aus, wenn die gedrückte STOP-Taste erkannt wurde.

OUT: Equal-Flag gesetzt, wenn STOP-Taste gedrückt war.

GETIN = $ffe4

Holen eines Zeichens vom aktuellen Eingabegerät

IN: –

Wie BASIN ist auch GETIN eine Eingaberoutine für das Holen von Daten von einem eventuell durch CHKIN oder CKOUT gesetzten Eingabegerät. Von BASIN unterscheidet sich GETIN in der Behandlung der Tastatur und der RS232. Für uns ist nur die Tastatur

interessant. GETIN geht nicht wie BASIN den Umweg über eine Zeichenausgabe auf den Bildschirm bei blinkendem Cursor. GETIN holt direkt ein Zeichen aus dem Tastaturpuffer beziehungsweise aus einem Funktionstasten-String. Ist weder eine Funktionstaste abzuarbeiten noch ein Zeichen im Puffer, liefert GETIN ein Nullbyte zurück.
OUT: A = geholtes Zeichen
COM: Achtung! Register werden nicht gerettet. Das Carry-Flag ist beim rts immer clear.

CLALL = $ffe7

Anzahl offener Files gleich Null setzen, CLRCH
IN: –
CLALL löscht alle Tabelleneinträge offener Files, indem die Anzahl offener Files auf Null gesetzt wird. Anschließend wird ein CLRCH durchgeführt und so die Standardwerte für Ein- und Ausgabegerät gesetzt. CLALL schließt keine Files (auf Diskette beispielsweise), da kein ordentliches CLOSE durchgeführt wird.
OUT: –

UDTIM = $ffea

Erhöhen der System-Uhrzeit um $^1/_{60}$ Sekunde
IN: –
UDTIM ist dieselbe Routine, die auch während des Interrupts für das Hochzählen der Uhrzeit zuständig ist. Eventuell kann man UDTIM benutzen, wenn der Interrupt-Vektor auf eine eigene Interrupt-Routine gerichtet wird.
OUT: –

SCRORG = $ffed

Liefert Größe des aktuellen Bildschirms (Bildschirmfensters)
IN: –
SCRORG liefert sowohl die Breite des aktuellen Gesamtbildschirms als auch Breite und Höhe des aktuellen Bildschirmfensters zurück. Die Höhe berechnet sich aus der Differenz zwischen unterem und oberem Fensterrand (SCBOT – SCTOP), die Breite entsprechend aus rechtem minus linkem Fensterrand (SCRT – SCLF). Die Breite des Gesamtschirms ist in LINES zu finden.
OUT: A = Breite Gesamtschirm
X = Breite des Fensters
Y = Höhe des Fensters

PLOT = $fff0

Setzen und Holen der Cursor-Position
IN: sec, clc
X = relative Zeile
Y = relative Spalte

Der Aufruf von PLOT mit gesetztem Carry führt zum Holen der Cursor-Position. Sowohl beim Holen als auch beim Setzen der Cursor-Position muß beachtet werden, daß die in X- und Y-Register übergebene Position relativ zu den Fenstergrenzen angegeben wird. Dies bedeutet, daß PLOT beim Setzen der Cursor-Position zu X den oberen Fensterrand addiert und zu Y den linken Fensterrand. Beim Holen der Cursor-Position werden die Ränder entsprechend von der tatsächlichen Position subtrahiert und in X und Y übergeben. Durch das Setzen des Cursors mit PLOT wird gleichzeitig der Start der Eingabezeile auf die Cursor-Position gesetzt. Überschreitet die angegebene Zeile oder Spalte die untere oder rechte Fenstergrenze, liefert die Routine ein gesetztes Carry als Fehler-Flag zurück.

OUT: X = relative Zeile
Y = relative Spalte
COM: Carry ist Fehler-Flag.

IOBASE = $fff3

Liefert Adresse des I/O-Bereiches zurück
IN: –
Für manche Anwenderprogramme, die auf mehreren Computertypen arbeiten sollen, ist es nützlich, die Adresse des jeweiligen I/O-Bereiches anhand der IOBASE-Routine erfragen zu können. Für normale Aufgaben wird diese Routine nicht benötigt.
OUT: X = $00
Y = $d0

4 Die Kernal-Vektoren in der erweiterten Zeropage

Zunächst zur Frage, was Vektoren eigentlich sind.
Es gibt zwei Arten von Vektoren in der erweiterten Zeropage. Die erste, die schon bei der Beschreibung der Tastaturroutinen auftauchte, stellt einen Zeiger auf eine Tabelle dar. Das Betriebssystem holt zum Beispiel seine Decodierinformationen aus einer Tabelle, die in einem Vektor definiert wird. Diese Vorgehensweise hat für den Programmierer den großen Vorteil, daß er die Lage der Tabelle verändern kann, denn der Vektor, aus dem das Kernal liest, liegt ja im RAM.
Die zweite Art von Vektoren besteht aus sogenannten Sprungvektoren. Sprungvektoren sind in einige wichtige Kernal-Routinen in der Form eines indirekten Sprungs »jmp (...)« über den Vektor eingebaut. Durch eine Verlegung des Sprungvektors auf eine eigene Routine kann man eigene Anwendungen sehr bequem in die Kernal-Routinen einschleifen, wie auch später noch an Beispielen gezeigt wird.
Am Beispiel des Vektors für die Tastaturauswertung soll einmal gezeigt werden, wie ein Vektor innerhalb einer Kernal-Routine verwendet wird:

```
$c5db    pla
         lda $d4      ; Tastencode
         jmp ($33a)   ; Sprung zu der Adresse, die im Vektor
                        $33a/ $33b angegeben ist
                        normalerweise = $c5e1
$c5e1    cmp #$57     ; normale Routine: Tastencode auswerten
```

Von den vorhandenen Vektoren sollen in diesem Abschnitt nur die Sprungvektoren behandelt werden. An Tabellenvektoren existieren in der erweiterten Zeropage lediglich

die Zeiger auf die verschiedenen Decodiertabellen. Der Umgang mit diesen Vektoren müßte aber nach Lektüre des Abschnitts über die Tastatur und die Tastaturroutinen klar sein.
Die Sprungvektoren möchte ich nicht in der Reihenfolge behandeln wie sie im RAM stehen, sondern nach Sachgebieten geordnet.

4.1 Vektoren des Kernal-Editors ($334 - $33c)

Innerhalb des Kernal-Editors sind, außer den Tastatur-Tabellenvektoren, fünf weitere Vektoren, Sprungvektoren, zu finden. Drei von ihnen beziehen sich auf die Zeichenausgabe, die restlichen zwei lassen einen Eingriff in die Tastaturroutinen zu. Die Vektoren sind Normalwerte in Klammern:

IPRCTRL	= **$334/$335**	Ausgabe eines Zeichens mit Control ($c7b9)
IPRSHFT	= **$336/$337**	Ausgabe mit Shift ($c805)
IPRESC	= **$338/$339**	Ausgabe mit Escape ($c9c1)
IKEYLOG	= **$33a/$33b**	Ergebnis der Tastaturmatrix-Abfrage auswerten ($c5e1)
IKEYSTO	= **$33c/$33d**	Tastencode in Tastaturpuffer schreiben (c6ad)

4.1.1 Vektoren zur Zeichenausgabe

Gehen wir einmal von der Voraussetzung aus, die Vektoren seien auf eine von uns geschriebene Routine gerichtet worden, so liegt beim Einsprung in unsere Routine folgender Sachverhalt vor:
In die Routine sind wir gelangt, indem ein Zeichen auf den Bildschirm ausgegeben werden mußte. Je nach Vektor war dieses Zeichen im Bereich der Control-Codes (ASCII-Wert von 0 bis $1f = 31) ein geshiftetes Zeichen (ASCII größer als $7f = 127), oder das vorausgegangene ausgegebene Zeichen war ein ESCAPE (ASCII $1b = 27). Das aktuell auszugebende Zeichen befindet sich zum Zeitpunkt des Eintritts in unsere Routine im Akkumulator, alle anderen benötigten Werte befinden sich in der Zeropage (siehe Beschreibung der Bildschirmausgabe).
Zur Rückkehr von unserer Routine zur Kernal-Routine PRINT genügt ein »rts«. Man beachte hier aber, daß das Kernal-ROM beim Rücksprung eingeschaltet sein muß. Dies nur für den Fall, daß unsere Routine zum Beispiel unter dem Kernal-ROM selbst liegt. In diesem Fall braucht man eine Zwischenstation außerhalb des Adreßbereiches, den das Kernal-ROM einnimmt, um die eigene Routine mit JSRFAR anzuspringen. Für eine

solche Zwischenstation bietet sich der Bereich $1300 bis $1c00 an, der normalerweise unbenutzt ist.
Als erstes Beispiel für die Benutzung der Vektoren soll der Vektor IPRCTRL herhalten. Es sollen die Rahmen- und die Hintergrundfarbe des VIC-Bildschirms mittels zweier Control-Codes umgeschaltet werden. Mit Control + u soll die Rahmenfarbe jeweils zyklisch erhöht werden, mit Control + v dasselbe für die Hintergrundfarbe:

```
          .base $1300

.define iprctrl = $334  ; Vektor für Ctrl-Ausgabe
.define old     = $c7b9 ; alter Vektorinhalt
.define vic     = $d000 ; Basisadresse des VIC
.define cr      = $ff00 ; Konfigurationsregister

init      lda #<(own)   ; Initialisierung des Vektors mit
          sta iprctrl   ; der Adresse der eigenen Routine
          lda # >(own)
          sta iprctrl+1
          rts

ende      lda #<(old)   ; Abschalten der Erweiterung durch
          sta iprctrl   ; Einschreiben der alten
          lda #>(old)   ; Adresse
          sta iprctrl
          rts

own       cmp #21       ; Code von CTRL + u
          beq rand      ; ja: Sprung
          cmp #22       ; Code von CTRL + v
          beq grund     ; ja: Sprung
          jmp old       ; sonst zur alten Routine

rand      ldy #0        ; Offset für Rahmenregister
          .byte $2c     ; Bit-Befehl: Überlesen von 2 Byte
grund     ldy #1        ; Offset Hintergrundfarbe 0
          da #0         ; I/O sichtbar machen
          sta cr
          ldx vic+32,y  ; Farbenregister lesen
          inx           ; und um 1 erhöhen
          txa           ; in den Akku und
          and #15       ; modulo 16, da nur 16 Farben
          sta vic+32,y  ; wieder abspeichern
          rts           ; fertig
```

Ich glaube, die Wirkungsweise dieser kleinen Routine dürfte klar sein. Nach der Initialisierung mit sys dec("1300") stehen die neuen Control-Codes zur Verfügung. Die Besonderheit, daß die VIC-Register für Rahmen- und Hintergrundfarbe hintereinander liegen, macht es möglich, diese Register y-indiziert zu lesen, was das Programm kürzer macht. Das seltsame .byte $2c bedarf vielleicht noch einer Erklärung. Das $2c ist der Befehlscode des Bit-Befehls. Er bewirkt, daß die folgenden 2 Byte als Operanden des Bit-Befehls angesehen werden, wenn der Prozessor vom Punkt »rand« aus die Routine abarbeitet. Dadurch überliest der Prozessor das »ldy #1«, der Bit-Befehl selbst ändert aber keine Registerinhalte. Dieser kleine Trick wird übrigens sehr häufig angewandt, da man sich durch seine Verwendung eine Verzweigung erspart, die zum einen 1 Byte länger, zum anderen aber auch langsamer ist:

```
ldy #0          =        ldy #0
.byte $2c                beq marke
ldy #1                   ldy #1
                   marke ...
```

Das gleiche Spiel geht auch mit dem Bit-Opcode $24 (36), der die Variante des Bit-Befehls mit Zeropage-Adressierung ist. Da bei diesem Befehl nur 1 Byte als Operand oder Adressierung erwartet wird, lassen sich damit Konstruktionen wie die folgende bilden:

```
clc             =        clc
.byte $24                bcc marke
sec                      sec
                   marke
```

Unser Beispielprogramm läßt sich natürlich auch auf die gleichzeitige Behandlung der VDC-Farben abstimmen, wie wär's zur Übung damit?

Der Vektor IPRESC wird ganz analog zu IPRCTRL behandelt. Als Beispiel soll wieder ein kleines Programm dienen, das auf die Ausgabe von ESCAPE + »1« die Schriftfarbe des gesamten VIC-Bildschirms ebenso wie im vorigen Beispiel zyklisch ändert. Als Ausgangswert für die Farbe wird die Farbe des Zeichens an der HOME-Position genommen:

```
          .base $1300

.define ipresc = $338      ; Vektor für Ausgabe von ESC-Sequ.
.define old    = $c9c1     ; alter Inhalt des Vektors
.define cram   = $d800     ; Startadresse des VIC-Farb-RAMs
.define hip    = $da       ; Hilfszeiger 3 Byte
```

```
init    lda #<(own)     ; eigene Routine initialisieren
        sta ipresc
        lda #>(own)
        sta ipresc+1
        rts
ende    lda # <(old)    ; Erweiterung ausschalten
        sta ipresc
        lda #>(old)
        sta ipresc+1
        rts

own     cmp #"1"        ; auszugebendes Zeichen ASCII "1"?
        beq farbe       ; ja: Sprung
        jmp old         ; sonst zur alten Routine

farbe   lda #0
        sta cr          ; I/O sichtbar machen
        ldx cram        ; Farbe des Zeichens an HOME-Pos.
        inx             ; um 1 erhöhen
        txa             ; in den Akku
        and #15         ; modulo 16, weil nur 16 Farben
        sta hip+2       ; in Zwischenspeicher
        lda #<(cram)    ; Adresse des Farb-RAMs
        ldx #>(cram)    ; in einen
        sta hip         ; Hilfszeiger bringen
        stx hip+1
        ldx # 25        ; 25 Zeilen
m2      lda hip+2       ; Farbcode zurückholen
        ldy #0          ; bei Spalte 0 beginnend
m1      sta (hip),y     ; neue Farbe in Farb-RAM schreiben
        iny             ; nächste Spalte
        cpy #40         ; schon 40 Spalten?
        bne m1          ; nein: Schleife
        dex             ; Zeilenzähler-1
        beq fertig      ; =0: Ende der Routine
        clc             ; Carry löschen für Addition
        tya             ; hip+40
        adc hip         ; ist neuer Zeiger
        sta hip         ; auf nächste Zeile
        bcc m2          ; kein Überlauf: Schleife
        inc hip+1       ; Highbyte erhöhen
```

```
            bcs m2              ; immer Sprung

  fertig    rts
```

Auch dieses kleine Beispielprogramm läßt sich relativ einfach an die Behandlung des VDC-Bildschirms anpassen.
Den letzten Vektor, IPRSHFT, behandelt man auf die gleiche Weise wie die beiden vorangegangenen, ich möchte mir deshalb ein Beispiel hierzu ersparen.

4.1.2 Die Tastaturvektoren

In den Tastaturroutinen liegen zwei Vektoren, die beide mit Beispielen versehen werden sollen. Die Vektoren ermöglichen die Beeinflussung der Tastaturbehandlung, ohne daß dazu in den Interrupt eingegriffen werden muß – ein großes Plus gegenüber dem C64.

4.1.2.1 Der Vektor IKEYLOG ($33a)

Der erste Vektor, IKEYLOG, liegt an der Schnittstelle zwischen den weiter oben (Kapitel 3) beschriebenen Routinen KEYSCN und KEYLOG, also zwischen der eigentlichen Abfrage der Tastaturmatrix und der Auswertung des Ergebnisses dieser Abfrage.
Legen wir den IKEYLOG-Vektor auf eine eigene Routine, haben wir beim Einsprung in die Routine folgende Parameter zur Verfügung:

- der Akkumulator enthält die Tastennummer. Diese ist auch in SFDX = $d4 in der Zeropage wiederzufinden.
- das aktuelle Shift-Muster ist in SHFFLG = $d3 zu finden.

Da die KEYSCN- und die KEYLOG-Routine während des Interrupts aufgerufen werden, muß man bei der Veränderung des Vektorinhaltes darauf achtgeben, während der Änderung keine Interrupts zuzulassen. Daß der KEYSCN-Aufruf beim Interrupt erfolgt, gibt uns außerdem die Chance, Aktionen, die durch eine eigene Routine ausgelöst werden sollen, praktisch jederzeit und augenblicklich wirken zu lassen. Wer schon einmal die Demo-Diskette der 1571-Floppy in der Hand gehabt hat, wird ein Beispiel kennen, das auf diese Weise arbeitet: Die Aktivierung beziehungsweise Desaktivierung des DOS-SHELL-Programms über die F1-Taste.
Das erste Beispiel zum IKEYLOG-Vektor soll nach dieser Art verfahren. Das durch Tastendruck zu aktivierende Programm besteht aus dem Anzeigen des Fehlerkanals der Floppy (Gerät 8) bei Betätigung der F1-Taste. Gleichzeitig wird etwas davon deutlich, was bei der Programmierung von Routinen zu beachten ist, die während des Interrupts ablaufen sollen:

```
          .base $1300

.define ikeylog = $33a   ; Vektor für Tastaturauswertung
.define old     = $c5e1  ; alter Inhalt des Vektors
.define talk    = $ffb4  ; TALK senden
.define sectlk  = $ff96  ; Sekundäradresse zu TALK senden
.define untalk  = $ffab  ; UNTALK-Befehl senden
.define iecin   = $ffa5  ; Byte vom IEC-Bus holen
.define bsout   = $ffd2  ; Zeichen ausgeben (auf Bildschirm)
.define status  = $90    ; Statusbyte

init      sei            ; Interrupts verhindern
          lda #<(own)    ; Vektor auf eigene Routine
          ldx #>(own)    ; richten
          jmp ma1

ende      sei
          lda #<(old)    ; Vektor auf alten Wert
          ldx #>(old)    ; bringen
ma1       sta ikeylog
          stx ikeylog+1
          cli            ; Interrupts wieder erlauben
          rts

own       cmp #4         ; Tastennummer von F1?
          beq start      ; ja: Sprung
          jmp old        ; sonst zur alten Routine

start     lda flag       ; Flag für Routine aktuell benutzt
          bne ma2        ; wird gerade bearbeitet: rts
          inc flag       ; Flag setzen "Routine läuft"
          sta status     ; STATUS löschen
          lda #8         ; Gerät 8 adressieren
          jsr talk       ; Daten senden
          lda #15+$60    ; Kanalnummer 15+Zusatz normale
          jsr sectlk     ; Sekundäradresse senden
ma3       jsr iecin      ; Byte vom Bus holen
          jsr bsout      ; und ausgeben
          bit status     ; End of File?
          bvc ma3        ; nein: Schleife
          jsr untalk     ; ja: Gerät abhängen
          dec flag       ; Flag wieder auf Null (Routine
ma2       rts            ; frei) und fertig

flag      .byte 0        ; Flag mit Startwert 0
```

Auch diese Routine müßte eigentlich noch verbessert werden. Es fehlt zum Beispiel eine Fehlerabfrage, ob das Gerät 8 antwortet, und die Berücksichtigung, ob der Cursor bei der Ausgabe abgeschaltet ist: er bleibt nämlich manchmal noch stehen. Als Beispiel soll die Routine ja auch nur eine Anregung sein.

Die Konstruktion mit einem Flag, das anzeigt, ob das Programm gerade läuft oder nicht, erscheint sicher auf den ersten Blick etwas merkwürdig. Man muß sich allerdings vor Augen halten, daß jede $1/60$ Sekunde ein neuer Interrupt erfolgt, der unser Programm unterbrechen wird. Ist bei einer solchen Unterbrechung die F1-Taste noch gedrückt, was sehr leicht der Fall sein kann, führt dies ohne das Flag zu einem doppelten Aufruf der Routine, was im Endeffekt zu einem heillosen Durcheinander führt, das der Prozessor schließlich mit einem Absturz quittiert.

Das zweite Beispiel zum IKEYLOG-Vektor läuft ebenfalls während des Interrupts ab, hier sind jedoch nicht die Schwierigkeiten wie beim vorigen Beispiel zu erwarten, da die auszuführende Routine in diesem Fall nicht länger als die Zeit zwischen zwei Interrupts dauert. Ziel des Beispiels soll es sein, einen weiteren Satz von Funktionstasten zu gewinnen. Sie sollen auf den Tasten F1 bis F7 liegen und durch gleichzeitiges Betätigen der Control- und einer der Funktionstasten erreicht werden. Jede der so gewonnenen vier neuen Funktionstasten soll ein Tasten-String zugeordnet werden, der bei Betätigung der Taste wie sonst auch auf dem Bildschirm erscheint.
Zum besseren Verständnis des Beispiels möchte ich wieder auf das Kapitel 3.1 über die Tastatur hinweisen, in dem die Funktionsweise der Tastaturabfrage eingehend erläutert ist.
Ausgangsüberlegung für die folgende Routine ist, daß der Puffer für die Tasten-Strings (KEYSTR = $100a) normalerweise nicht vollständig gefüllt ist. Es wird sich in der Regel noch genug Platz im Puffer finden, in dem man eigene Tasten-Strings unterbringen kann. Durch Prüfung des aktuellen Shift-Musters wird festgestellt, ob die Control-Taste gedrückt ist. Ist dies der Fall und wird zusammen mit Control eine der Funktionstasten betätigt, so wird der der Taste zugeordnete Tasten-String in den Puffer für Tasten-Strings kopiert. Die Länge des Strings und seine Adresse innerhalb des Puffers werden an den zuständigen Stellen in der Zeropage abgelegt.

```
              .base $1300

    .define kyndx   =  $d1     ; Länge des Tasten-Strings
    .define kyidx   =  $d2     ; Index innerhalb KEYSTR
    .define shfflg  =  $d3     ; aktuelles Shift-Muster
    .define ikeylog =  $33a    ; Vektor Tastaturauswertung
    .define old     =  $c5e1   ; alter Inhalt des Vektors
    .define delay   =  $a24    ; Zähler Repeat-Frequenz
    .define sdelay  =  $a25    ; Ansprechverzögerung des Repeat
    .define keystr  =  $100a   ; Puffer für Tasten-Strings
```

```
init    sei                 ; Interrupts verhindern
        lda #<(own)         ; Vektor auf eigene
        ldx #>(own)         ; Routine richten
        jmp ma1

ende    sei
        lda #<(old)         ; alten Zustand
        ldx #>(old)         ; wiederherstellen
ma1     sta ikeylog
        stx ikeylog+1
        cli                 ; Interrupt wieder zulassen
        rts

own     ldx shfflg          ; Shift-Muster
        cpx #%0000'0100     ; gleich Control?
        beq own1            ; ja: weiter
not     jmp old             ; sonst: zur alten Routine

own1    cmp #3              ; Tastennummer kleiner als 3?
        bcc not             ; ja: keine Funktionstaste
        cmp #7              ; größer/gleich 7?
        bcs not             ; ja: keine Funktionstaste

        lsr sdelay          ; Wiederholungsverzögerung
        bne back            ; nicht abgel.: keine Aktion
        ldx delay           ; Wiederholungsfrequenz-Zähler
        beq own2            ; abgelaufen: Aktion
        dec delay           ; sonst: herunterzählen
        bne back            ; und keine Aktion

own2    sec                 ; Tastennummer - 3
        sbc #3
        tax                 ; als Zeiger in die Tabelle
        lda len,x           ; der String-Längen, String-
        sta kyndx           ; Länge in den zug. Zeiger
        ldy #0              ; Offset in String-Tabelle
        txa                 ; war es der erste String?
        beq ueber           ; ja: String-Suche übergehen

m2      lda text,y          ; Suchen eines Nullbytes in der
        beq m1              ; String-Tabelle, gefunden: Sprung
```

```
m3      iny             ; sonst: nächstes Zeichen
        bne m2          ; immer Sprung

m1      dex             ; paßt String zur Nummer?
        bne m3          ; nein: Sprung
        iny             ; ja: Zeiger hinter das Nullbyte

ueber   lda text,y      ; String-Zeichen in KEYSTR
        sta keystr+80,x ; an freie Position übertragen
        beq m4          ; bis zum Nullbyte
        inx             ; nächstes Zeichen
        iny
        bne m4          ; immer Sprung

m4      lda #80-10      ; Index des Strings in KEYSTR
        sta kyidx       ; in die Zeropage
        lda #8          ; Verzögerungszähler
        sta sdelay      ; initialisieren
        sta delay
back    rts             ; Ende

len     .byte 4,4,4,15  ; Tasten-String-Längen

text    .byte "dies",0  ; neue Tasten-Strings
        .byte "sind",0
        .byte "vier",0
        .byte "Funktionstasten",0
```

Die in dieser Routine verwendete Repeat-Verzögerung entspricht nicht der des Kernal-Editors, sie leistet aber das, was sie leisten soll. Ein Wort noch zu dem Index des Tasten-Strings innerhalb des String-Puffers. Der Index entspricht dem Lowbyte der Adresse des Tasten-Strings, allerdings muß die Zahl 10 abgezogen werden, da die ersten 10 Byte ab der Adresse $1000 für die Längenangaben der normalen Tasten-Strings reserviert sind. Daher rühren die Befehle 'lda #80-10' und 'sta kyidx'.

Daß eine Repeat-Verzögerung unbedingt notwendig ist, können Sie leicht selbst ausprobieren, indem Sie die zur Verzögerung verwendeten Befehle einmal weglassen. Es zeigt sich dann sehr deutlich, daß man eine Taste gar nicht so schnell loslassen kann, wie die Tastaturroutinen vom Prozessor abgearbeitet werden.

4.1.2.2 Der Vektor IKEYSTO ($33c, $c6ad)

Der Vektor IKEYSTO liegt am Ende der KEYLOG-Routine. Der Sprung über diesen Vektor ist an der Adresse $fd26 im Kernal-ROM zu finden. Der Normalinhalt des Vektors führt zur Adresse $c6ad im Kernal-Editor. Die Prozessorregister enthalten bei dem Sprung über IKEYSTO:

A = Tastencode der aktuellen Taste
X = aktuelles Shift-Muster ($d3)
Y = Tastennummer ($d5)

In der Fortsetzung von KEYLOG nach dem Sprung wird der im Akku übergebene Tastencode darauf überprüft, ob es sich um eine Funktionstaste handelt. Ist es eine Funktionstaste, werden Länge und Index des Tasten-Strings in KYNDX und KYIDX geschrieben, ansonsten wird der Tastencode in den Tastaturpuffer KEYBUF gebracht. Die beiden dafür notwendigen Routinen befinden sich bei:

$c6b7 Zeichen in Tastaturpuffer schreiben

und

$c6ca Parameter eines Tasten-Strings in die Zeropage bringen
(X = Nummer der F.-Taste-1)

Im allgemeinen bietet sich der IKEYSTO-Vektor eher für das Einschleusen eigener Routinen an als der IKEYLOG-Vektor, da man sich zum Beispiel die recht aufwendige Repeat-Logik sparen kann. Die Beispiele zum IKEYLOG-Vektor könnte man ohne weiteres auch auf den IKEYSTO-Vektor umschreiben.

Aus diesem Grund soll an dieser Stelle ein sehr einfaches Beispielprogramm genügen, um den Umgang mit dem IKEYSTO-Vektor kennenzulernen. Es soll einfach verhindert werden, daß überhaupt Tasten-Strings ausgegeben werden. Dazu genügt es, die Sprungadresse, die in dem Vektor enthalten ist, ein wenig zu verändern, so daß der Vergleich des Tastencodes im Akku mit den Codes der Funktionstasten unterbleibt:

```
                .base $1300

 .define ikeysto = $33c   ; Vektor Tastencode speichern
 .define old     = $c6ad  ; alter Inhalt des Vektors
 .define new     = $c6b7  ; dieselbe Routine ohne Vergleich
                            mit Funktionstasten-Codes
 init           sei       ; Interrupts verhindern
                lda #<(new) ; Vektor auf neue
```

```
            ldx #>(new) ; Routine legen
            bne ma1     ; immer Sprung

ende        sei         ; Interrupts verhindern
            lda #<(old) ; Vektor auf alten Stand
            lda #>(old) ; bringen
ma1         sta ikeysto
            stx ikeysto+1
            cli         ; Interrupts wieder zulassen
            rts         ; fertig
```

Als Ergebnis der Routine erhält man nun nicht mehr den Funktionstasten-String, sondern den normalen Tastencode der Funktionstaste im Tastaturpuffer.

4.2 Vektoren der I/O-Routinen

Unter dem Begriff »Vektoren der I/O-Routinen« möchte ich die Vektoren des Kernals zusammenfassen, die innerhalb der für die allgemeine I/O zuständigen Routinen liegen, als da sind:

IOPEN	= **$31a/ $31b**	in der OPEN-Routine ($efbd)
ICLOSE	= **$31c/ $31d**	in der CLOSE-Routine ($f188)
ICHKIN	= **$31e/ $31f**	bei CHKIN ($f106)
ICKOUT	= **$320/ $321**	bei CKOUT ($f14c)
ICLRCH	= **$322/ $323**	bei CLRCH ($f226)
IBASIN	= **$324/ $325**	bei BASIN ($ef06)
IBSOUT	= **$326/ $327**	bei BSOUT ($ef79)
IGETIN	= **$32a/ $32b**	bei GETIN ($eeeb)
ICLALL	= **$32c/ $32d**	bei CLALL ($f222)
ILOAD	= **$330/ $331**	bei LOAD ($f26c)
ISAVE	= **$332/ $333**	bei SAVE ($f54e)

Die Adressen in Klammern geben den normalen Inhalt der Vektoren wieder.
Wie man sieht, liegt in jeder wichtigen I/O-Routine gleichzeitig auch ein Sprung über einen Vektor in der erweiterten Zeropage. Dies ermöglicht die Anpassung der gesamten I/O an benutzerspezifische Anforderungen.
Alle diese indirekten Sprünge stehen gleich zu Anfang der jeweiligen Routine. Bei einem Einsprung in eine eigene Routine kann man deshalb Parameter in den Prozessorregistern

erwarten, die denen bei einem normalen Aufruf entsprechen. Man muß allerdings darauf achten, daß die eigene Routine, zum Beispiel was die Rettung von Registerinhalten anbelangt, dasselbe leistet wie die ursprüngliche Routine. Auch der Zustand des Carry-Flags bei der Rückkehr von einer selbstgeschriebenen Routine kann von Bedeutung sein. Siehe hierzu auch in Kapitel 3.5 die Beschreibung der Routinen, die zu den einzelnen Vektoren normalerweise gehören.
Kommen wir aber jetzt zu Beispielen der Anwendung dieser Vektoren. Es ist sicher nicht möglich, jeden einzelnen der Vektoren mit einem Beispiel zu versehen, das würde leicht den Rahmen dieses Buches übersteigen. Ich möchte mich deshalb auf zwei Beispiele beschränken.
Das erste Beispiel benutzt den ILOAD-Vektor und macht einen in der Kernal-Routine LOAD steckenden kleinen Fehler unschädlich, der verhindert, daß der BASIC-Befehl BOOT bei langsamem Laden eines Programms von einer 1541-Floppy wie geplant funktioniert.
Der Fehler ist darin begründet, daß in der langsamen Laderoutine vergessen wurde, die gelesene Startadresse des zu ladenden Maschinenprogramms in der Zeropage-Adresse SAL = $ac/ $ad abzulegen, wo sie später vom BOOT-Befehl dazu abgeholt wird, um das Programm mit JSRFAR zu starten.
Zur Fehlerbehebung öffnen wir einfach das zu ladende Programm-File, bevor die LOAD-Routine aufgerufen wird und lesen die Startadresse selbst.

Beispiel 1:

```
         .base $1300

  .define iload  = $330   ; LOAD-Vektor
  .define old    = $f26c  ; Adresse der LOAD-Routine
  .define sal    = $ac    ; Startzeiger
  .define status = $90    ; STATUS-Byte
  .define fnlen  = $b7    ; Länge des Filenamens
  .define fnadr  = $bb    ; Adresse des Filenamens
  .define fa     = $ba    ; aktuelle Gerätenummer
  .define sa     = $b9    ; aktuelle Sekundäradresse
  .define talk   = $ffb4  ; TALK-Befehl senden
  .define listen = $ffb1  ; LISTEN-Befehl
  .define sectlk = $ff96  ; Sekundäradresse für TALK
  .define seclst = $ff93  ; dasselbe für LISTEN
  .define untalk = $ffab  ; UNTALK-Befehl senden
  .define unlist = $ffae  ; UNLISTEN senden
  .define iecin  = $ffa5  ; Byte vom Bus holen
  .define iecout = $ffa8  ; Byte auf den Bus senden
```

```
init    lda #< (new)      ; LOAD-Vektor auf eigene
        ldx #> (new)      ; Routine richten
        jmp ma1

ende    lda #< (old)      ; alten Zustand wiederherstellen
        ldx #> (old)
ma1     sta iload
        stx iload+1
        rts

new     tax               ; Akku nicht 0: ist verify
        beq new1          ; Load: Sprung
        jmp old           ; Verify: zur alten Routine

; alle Parameter der LOAD-Routine sind durch SETBNK usw.
; schon gesetzt und stehen in der Zeropage zur Verfügung

new1    lda #0            ; STATUS löschen
        sta status
        lda fa            ; Geräteadresse
        jsr listen        ; LISTEN senden
        lda sa            ; Sekundäradresse
        ora #$f0          ; Zusatz für OPEN
        jsr seclst        ; Sekundäradresse senden
        ldy #0            ; Offset für Dateinamen senden
ma2     lda (fnadr),y     ; ein Zeichen des Namens senden
        jsr iecout
        iny               ; alle Zeichen?
        cpy fnlen
        bcc ma2           ; nein: Schleife
        jsr unlist        ; UNLISTEN senden

; die Datei ist jetzt offen und man kann die Startadresse
; lesen

        lda fa            ; Gerätenummer
        jsr talk          ; Gerät soll senden
        lda sa            ; Sekundäradresse
        jsr sectlk        ; senden
        jsr iecin         ; Byte vom Bus = Lowbyte des Starts
        sta sal           ; in den Startzeiger Lowbyte
```

```
jsr iecin           ; Highbyte
sta sal+1           ; in den Zeiger
jsr untalk          ; Gerät abhängen
lda #0              ; Kennzeichen für LOAD
jmp old             ; zur alten LOAD-Routine
```

Beachten Sie bitte auch bei diesem Beispielprogramm wie bei allen anderen Beispielen, daß sie nicht vollständig durchprogrammiert wiedergegeben sind, da dies einfach zu lang würde. So fehlt beim obigen Programm eine Fehlerabfrage (Prüfung von STATUS), oder auch die Überprüfung, ob überhaupt von Diskette oder aber von Kassette geladen werden soll. Ferner ist auch stillschweigend angenommen worden, daß der File-Name in Bank0 irgendwo »sichtbar« steht, was auch nicht der Fall sein muß. Allgemeiner müßten deshalb die Zeichen des File-Namens mit FETCH geholt werden.

Beispiel 2:

Dieses Beispiel kann sehr schön erweitert werden, so daß eine eigene Software-Drucker-Schnittstelle entsteht. Bekanntlich stimmt das Commodore-ASCII nicht in allen Punkten mit dem Standardzeichensatz des Standard-ASCII überein. Über den Vektor IBSOUT sollte es aber kein Problem sein, die Commodore-Zeichen in Standardzeichen umzuwandeln, wie sie von den meisten Druckern, so etwa von EPSON, benutzt werden.
Für unser Beispiel genügt das Prinzip der Ausgabe. Es sollen daher in dem folgenden kleinen Programm lediglich die kleinen Buchstaben des Commodore-ASCII (CBM-ASCII von 65 bis 90) in kleine Buchstaben des Standard-ASCII (Codes von 97 bis 122) überführt werden. Alles weitere soll Ihrer Phantasie und Ihrem Schaffensdrang überlassen bleiben.

```
          .base $1300

.define ibsout = $326   ; BSOUT-Vektor
.define old    = $ef79  ; altes Sprungziel
.define dflto  = $9a    ; aktuelles Ausgabegerät

init      lda #<(new) ; BSOUT-Vektor auf neue Routine
          ldx #>(new) ; richten
          jmp ma1

ende      lda #<(old) ; Standard
          ldx #>(old) ; wiederherstellen
ma1       sta ibsout
          stx ibsout
          rts
```

```
new     pha         ; auszugebendes Zeichen merken
        lda dflto   ; Ausgabegerät
        cmp #4      ; Drucker?
        beq is      ; ja: Sprung
        pla         ; nein: Zeichen vom Stack holen
isnot   jmp old     ; beim alten BSOUT weiter

is      pla         ; Zeichen vom Stack
        cmp #65     ; kleiner als 65?
        bcc isnot   ; ja: Sprung
        cmp #91     ; größer/gleich 90?
        bcs isnot   ; ja: keine Wandlung
        adc #22     ; sonst: Zeichen anpassen
        bcc isnot   ; und ausgeben
```

Auf die gleiche Art und Weise kann man auch die anderen CBM-Zeichen anpassen, so daß ein Drucker sie richtig interpretieren kann. Man könnte auch noch Spezialfunktionen einbauen, die zum Beispiel die Software-Schnittstelle in Abhängigkeit von der eingestellten Sekundäradresse unterschiedlich arbeiten läßt und was der Dinge mehr sind. Insgesamt ist dies bestimmt ein interessantes Gebiet, gerade auch für den Einsteiger in die Maschinensprache, zumal eine Hardware-Schnittstelle, die im Prinzip dasselbe macht, einiges Geld kostet.

Das gleiche Problem, nämlich, daß Daten von einem ASCII-Satz in einen anderen übertragen werden müssen, kann natürlich auch bei der Eingabe entstehen. Man denke etwa an den Datenempfang über ein Modem bei der Daten-Fernübertragung (DFÜ). Dies läßt sich dann mit umgekehrtem Vorzeichen über den Vektor IBASIN abwickeln.

Auch eine kleine RAM-Disk könnte man über die Verwendung der I/O-Vektoren zusammenbasteln. Man weist ihr zum Beispiel die Gerätenummer 20 zu und schreibt entsprechende Routinen, die ein Zeichen nicht ausgeben, sondern in RAM-Bank 1 abspeichern. OPEN, CLOSE usw. müßten angepaßt werden, damit keine Fehlermeldung wie »device not present« ausgegeben wird. Dies nur als Anregung.

4.3 Der STOP-Vektor (ISTOP, $328)

Der Vektor ISTOP führt normalerweise zur STOP-Routine des Kernals, die die gedrückte Stop-Taste abfragt. War die Stop-Taste beim Einsprung in die Routine gedrückt, übergibt sie bei der Rückkehr eine Null im Akkumulator. Siehe hierzu auch die Beschreibung der STOP-Routine in Kapitel 3.5. Der Standardwert für den Stop-Vektor beträgt $f66e, was zugleich die Adresse der STOP-Routine im ROM ist.

Das einzige Beispiel einer Anwendung des ISTOP-Vektors, das mir ad hoc einfällt, wäre, die Stop-Taste zu sperren, indem dafür gesorgt wird, daß der Akkumulator in einer eigenen Routine garantiert nicht 0 ist. Dies hat gleichzeitig den Effekt, daß ein BASIC-Programm nicht mehr mit Stop angehalten werden kann, was von Zeit zu Zeit sicher nützlich sein kann.

```
          .base $1300

.define istop  = $328   ; Stop-Vektor
.define old    = $f66e  ; Default-Wert des Vektors

init      lda #<(own)
          ldx #>(own)
          jmp ma1
ende      lda #<(old)
          ldx #>(old)
ma1       sta istop
          stx istop+1
          rts

own       lda #1        ; Akku ungleich 0 genügt schon
          rts
```

Da bei der NMI-Unterbrechung mittels RUN/STOP- und RESTORE-Taste ebenfalls die STOP-Routine zur Abfrage herhält, haben wir mit der Versetzung des ISTOP-Vektors gleichzeitig diesen Fall verhindert. Ein laufendes BASIC-Programm ist damit praktisch nur durch ein RESET zu stoppen.

4.4 Die Interrupt-Vektoren, der Systemvektor

Was ist eigentlich unter einem Interrupt zu verstehen? Diese Frage ist zunächst von Bedeutung.

Ein Interrupt ist ein Ereignis, das hardware- oder softwaremäßig hervorgerufen werden kann. Hardwaremäßig zum Beispiel durch einen Timer der CIAs, softwaremäßig nur durch den Prozessorbefehl »brk«.

Alle Interrupts führen dazu, daß der Prozessor, gleichgültig, was er gerade macht, mit der Arbeit innehält, sich den aktuellen Programmzähler und den Prozessorstatus auf dem Prozessorstack merkt, und daraufhin die Arbeit an der Stelle aufnimmt, die in einem Vektor am Ende des Speichers definiert ist.

Es gibt zwei solcher Vektoren, woraus wir schließen können, daß es auch zwei prinzipiell unterschiedliche Arten von Interrupts gibt.
Die erste Art von Interrupt ist dadurch gekennzeichnet, daß man sie mit dem Prozessorbefehl »sei« verhindern und mit »cli« zulassen kann. Diese Art nennt man maskierbare Interrupts. Der zuständige Vektor für diese Interrupts liegt bei

IRQ = $fffe/ $ffff

Die zweite Art von Interrupt heißt entsprechend, weil nicht auszuschalten, nicht maskierbare Interrupt (NMI). Beim Auftreten einer solchen Unterbrechung wird über den Vektor

NMI = $fffa/ $fffb

gesprungen, um zur NMI-Routine zu gelangen. Beim C128 ist wie schon beim C64 die RESTORE-Taste direkt mit der NMI-Leitung der 8502-CPU verbunden.
Beide Vektoren werden von der Hardware bestimmt, man hat auf die Lage der Vektoren deshalb keinen Einfluß. Dieses ist aber kein Manko, denn die IRQ- und die NMI-Routinen des Kernals enthalten beide Vektoren, die im RAM liegen. Wir können also auch in die Interrupts eingreifen.
Die IRQ- und die NMI-Routine nehmen uns vor dem Sprung über den zugehörigen RAM-Vektor sogar noch einige Arbeit ab, denn sie legen auch die Inhalte der Prozessorregister und die aktuelle Konfiguration beim Auftreten des Interrupts zu den schon vom Prozessor gesicherten Werten auf den Stack. Von dort können diese Werte beim Verlassen der Interrupt-Routine gegebenenfalls wieder zurückgeholt, und die Arbeit dort wieder aufgenommen werden, wo der Prozessor unterbrochen wurde.
Die IRQ-Routine unterscheidet außerdem anhand des BRK-Flags des Prozessor-Status-Registers, ob die Unterbrechung von der Hardware, beispielsweise durch den regelmäßigen Tastatur-Interrupt, oder von der Software (brk) stammt. Beide Fälle führen über unterschiedliche Vektoren im RAM.
Die Vektoren sind (Standardwerte in Klammern):

IIRQ	= **$314/ $315**	Interrupt-Vektor Hardware ($fa65)
IBRK	= **$316/ $317**	Interrupt durch brk ($b003 = Monitoreinsprung)
INMI	= **$318/ $319**	NMI-Vektor ($fa40)

Das ist sicher eine sehr komprimierte Darstellung der Interrupts, aber mehr war platzmäßig nicht drin. Ich hoffe, daß die meisten sowieso wissen, was ein Interrupt ist.
Wenden wir uns deshalb einem Beispiel zu, das erläutert, welche Anwendungen man mit Hilfe der Interrupt-Vektoren realisieren kann. Ich habe dazu den INMI-Vektor ausgesucht, über den gesprungen wird, wenn die RESTORE-Taste gedrückt wird. Normalerweise erfolgt in der NMI-Routine eine Abfrage auf die STOP-Taste, was zu dem bekannten

BASIC-Warmstart führt. Die folgende kleine Routine soll zusätzlich dazu den VIC-Bildschirm in ein Grafik- und ein Textfenster teilen, wobei die CBM-Taste (C=) zusammen mit RESTORE die Grenze zwischen den Fenstern nach unten verschiebt, die Control-Taste nach oben.

```
         .base $1300

.define inmi   = $318     ; NMI-Vektor im RAM
.define old    = $fa40    ; alter Inhalt des Vektors
.define graphm = $d8      ; Grafik-Flag
.define split  = $a34     ; Rasterzeile für IRQ
.define matrix = $f63d    ; Abfrage Tastaturmatrix, Reihe der STOP-Taste
.define stkey  = $91      ; Abfrageergebnis

init     lda #<(own)      ; eigene Routine initialisieren
         ldx #>(own)
         ldy #%0110'0000  ; Bildschirm auf Splitscreen
         sty graphm       ; einstellen
         bne ma1          ; immer: Sprung

ende     lda #<(old)      ; alten Vektor restaurieren
         ldx #>(old)
ma1      sta inmi
         stx inmi+1
         rts

own      jsr matrix       ; Tastaturmatrix-Abfrage
         lda stkey        ; gefundenen Code holen
         cmp #$df         ; CBM-Taste?
         bne ma2          ; nein: Sprung
         inc split        ; Grafikfenster vergrößern
         jmp ma3
ma2      cmp #$fb         ; Control-Taste?
         bne ma3          ; nein: Sprung
         dec split        ; Grafikfenster verkleinern
ma3      jmp old          ; alte NMI-Routine abarbeiten
```

Interessant ist sicher noch die Kernal-Routine »matrix«, die eine spezielle Tastaturmatrix-Abfrage durchführt, wobei nur die Tastenreihe geprüft wird, in der sich die STOP-Taste befindet. Da in dieser Reihe außer STOP noch andere nützliche Tasten sind, kann man diese Tastaturabfrage sicher gut gebrauchen. Welche Tasten von der Abfrage erfaßt werden, können Sie leicht von BASIC aus nachprüfen:

```
10 print peek(dec("91")):goto10
```

Der Code von STOP in der Abfrage ist $7f, er läßt sich natürlich nicht aus einem BASIC-Programm heraus ermitteln.

Innerhalb der NMI-Routine des Kernals findet sich noch ein weiterer Vektor wieder, der Systemvektor.

SYSTEM = $a00/ $a01 System-RESTART-Vektor ($4003)

Dieser Vektor enthält ganz einfach die Startadresse des System-Maschinenprogramms, im Normalfall also die Warmstart-Adresse des BASIC. Durch Verlegen dieses Vektors auf den Warmstart eines eigenen Maschinenprogramms erhält man an dieser Stelle einen Schutz gegen die RUN/STOP-+RESTORE-Unterbrechung.

4.5 RESET

Der RESET gehört eigentlich nicht zu den Interrupts, obwohl das Drücken der RESET-Taste natürlich auch eine Unterbrechung ist. Zudem liegt auch kein Vektor in der erweiterten Zeropage, der für ein RESET zuständig wäre. Da man aber irgendwo zumindest anreißen muß, was bei einem RESET beeinflußt werden kann, hat das Zurücksetzen des Computers in den Einschaltzustand an dieser Stelle ein eigenes Kapitel erhalten.

Der RESET-Knopf wirkt direkt auf den Prozessor. Er aktiviert eine RESET-Leitung, die den Prozessor veranlaßt, die Arbeit an der Adresse aufzunehmen, die in

RESET = $fffc/ $fffd

angegeben ist. Dies ist im Normalfall die Einsprung-Adresse der Kernal-RESET-Routine $ff3d.
Im Routinenkopf bei $ff3d wird allerdings lediglich das Konfigurationsregister mit Null geladen, so daß alle ROMs sichtbar werden und sodann zur eigentlichen RESET-Routine bei $e000 zu Beginn des Kernals weiterspringen.
Bei $e000 findet dann die wirkliche Initialisierung des gesamten Systems statt. Alle Einzel-Inits möchte ich hier nicht aufzählen. Man kann jedoch den Ablauf des RESET beeinflussen und das ist wieder interessant.
Zunächst wird auch innerhalb der RESET-Routine dieselbe Tastaturmatrix-Abfrage wie beim NMI-Interrupt aufgerufen. Für das RESET sind nur die Fälle interessant, wo die

STOP- oder die CBM-Taste während des Betätigens des RESET-Knopfes gedrückt gehalten wird.
Wurde die STOP-Taste gedrückt, verzweigt der RESET direkt in den Monitor, bei der CBM-Taste wird in den 64er-Modus übergegangen. In beiden Fällen wird der BOOT-Versuch, der normalerweise erfolgt, umgangen.
Wurde keine der beiden Tasten betätigt, sind nur noch die Fälle zu unterscheiden, bei denen der BOOT-Versuch erfolgreich abgelaufen ist oder nicht. Ein erfolgreiches Booten zieht natürlich in der Regel den Start des gebooteten Programms nach sich. Ist der Versuch negativ abgelaufen, wird der SYSTEM-Vektor angesprungen, der schon beim NMI angesprochen wurde.

Soweit der normale Ablauf eines RESET. Es gibt aber noch eine Möglichkeit, eine eigene RESET-Routine einzuschleusen.
Das Betriebssystem überprüft nämlich ziemlich zu Beginn der RESET-Routine die Adressen $fff5 bis $fff7 in der RAM-Bank 1. Liegen hier die Zeichen »cbm«, so werden die folgenden 2 Byte in $fff8 und $fff9 als Sprungziel übernommen, die RESET-Routine setzt sich an dieser Adresse in der RAM-Bank 0 fort.

Die Abfrage auf »cbm« in Bank 1 geschieht in der Routine $e1f0 und ist dort näher nachzulesen. Die Routine wird von RESET in Form eines Unterprogramms aufgerufen.
Durch Verwendung des RAM-RESET-Vektors der Bank 1 kann man relativ leicht eigene Maschinenprogramme »resetfest« machen. Das Stoppen eines BASIC-Programms durch RESET ist wohl nicht aufzuhalten, zumindest kann man in einem solchen Fall aber dafür sorgen, daß das Programm gelöscht wird oder der Computer nur noch durch ein Ausschalten wieder funktionstüchtig wird.
Hierzu genügt es ja schon, den RAM-RESET-Vektor so zu verbiegen, daß der Prozessor »auf der Stelle tritt«:

```
bank 1 : poke dec("fff8"), dec("21")
```

4.6 Der Monitor-Erweiterungsvektor EXMON ($32e, $b003)

Man kann geteilter Meinung sein, ob der eingebaute Monitor des C128 nun zum Betriebssystem oder zum BASIC gezählt werden soll, wie etwa der Sprite-Editor. Da der Monitor-Erweiterungsvektor aber inmitten anderer Kernal-Vektoren zu finden ist, soll er hier abgehandelt werden.

Der EXMON-Vektor erlaubt es, in das Monitorprogramm eigene Befehle einzuschleifen. Er weist auf den Anfang der Befehlserkennung des Monitors.
Wie BASIC auch, holt sich der Monitor eine Befehlszeile in den Eingabepuffer ab $200. Die Befehlserkennung beginnt damit, das erste Zeichen im Eingabepuffer mit den vorhandenen Befehlskürzeln zu vergleichen und bei Gleichheit zur entsprechenden Routine zu verzweigen.
Ich möchte als Beispiel für eine Monitorerweiterung einmal die Möglichkeit schaffen, während der Arbeit mit dem Monitor ein Druckerprotokoll auszugeben. Jedes Zeichen, das auf den Bildschirm ausgegeben wird, soll auch auf dem Drucker erscheinen.
Der Befehl »p« soll das Protokoll einschalten, der Befehl »o« soll es wieder ausschalten. Der Einfachheit halber bediene ich mich des BSOUT-Vektors, um sowohl eine Ausgabe auf den Schirm als auch auf den Drucker zu erzeugen:

```
          .base $1300

.define exmon  = $32e     ; Monitor-Extension-Vektor
.define oldmn  = $b006    ; alte Einsprungadresse
.define ibsout = $326     ; BSOUT-Vektor
.define oldout = $ef79    ; BSOUT ohne indirekten Sprung
.define setpar = $ffba    ; File-Parameter setzen
.define open   = $ffc0    ; File öffnen
.define close  = $ffc3    ; File schließen
.define ckout  = $ffc9    ; Ausgabekanal schalten
.define clrch  = $ffcc    ; Ein- und Ausgabe auf Default
.define monin  = $b08b    ; Einsprung in den Monitor:
                            Eingabezeile holen
.define setnam = $ffbd    ; File-Namen setzen

init      lda #< (own)    ; EXMON-Vektor auf eigene
          ldx #> (own)    ; Befehlserkennung
          sta exmon
          stx exmon+1
          rts

own       cmp #"p"        ; Protokoll ein?
          beq proton      ; ja: Sprung
          cmp #"o"        ; Protokoll aus?
          beq off         ; ja: Sprung
          jmp oldmn       ; weitere Befehle durch alte Erk.

; Protokoll einschalten
```

```
proton  lda #<(out)     ; BSOUT-Vektor auf die eigene
        ldx #>(out)     ; Ausgaberoutine legen
        sta ibsout
        sta ibsout+1
        lda #0          ; logische File-Nummer 0
        ldx #4          ; Gerät 4 = Drucker
        ldy #1          ; Sekundäradresse
        jsr setpar      ; Parameter setzen
        lda #0          ; File-Namen-Länge = 0
        jsr setnam      ; Parameter kein Name
        jsr open        ; Drucker öffnen
        jmp monin       ; zum Monitor

; Protokoll ausschalten

off     lda #0          ; logische File-Nummer 0
        jsr close       ; File schließen
        lda #<(oldout); alten BSOUT-Vektor
        ldx #>(oldout); wiederherstellen
        sta ibsout
        stx ibsout+1
        jmp monin       ; zurück zum Monitor

; neue Ausgaberoutine

out     jsr oldout      ; Zeichen auf Bildschirm ausg.
        sta zeichen     ; und zwischenspeichern
        txa             ; Register retten
        pha
        tya
        pha
        ldx #0          ; logische Datei 0 auf Ausgabe
        jsr ckout
        lda zeichen     ; Zeichen auf Drucker
        jsr oldout      ; ausgeben
        jsr clrch       ; Ausgabe auf Default
        pla             ; Register zurück
        tay
        pla
        tax
        lda zeichen     ; Akku restaurieren
```

```
          clc
          rts              ; fertig

zeichen   .byte 0          ; Zwischenspeicher
```

Für weitere Erweiterungen wäre schon ein ROM-Listing notwendig, um die Funktionsweise der einzelnen Monitorroutinen, die Parameter holen usw., zu analysieren. Den Besitz eines solchen Listings möchte ich aber nicht unbedingt voraussetzen. Auf Dauer wird die Programmierung des C128 jedoch durch ein ROM-Listing sehr viel einfacher, weshalb ich die Anschaffung nur empfehlen kann.

5
Der BASIC-Interpreter

Der BASIC-Interpreter belegt im C128 etwa 24 KByte, verteilt auf zwei ROMs. Das erste ROM liegt im Adreßbereich von $4000 bis $7fff, das zweite von $8000 bis $bfff. Im zweiten ROM findet sich ab Adresse $b000 auch der C128-Bordmonitor wieder.
Man könnte nun fragen, warum man sich als Programmierer in Maschinensprache für den Interpreter interessieren sollte. Die Antwort auf diese Frage ist aber recht einfach: Zunächst einmal kann man viele der BASIC-ROM-Routinen auch in eigenen Maschinenprogrammen gut gebrauchen. Die BASIC-ROMs können uns also als eine ziemlich große Bibliothek an nützlichen Unterprogrammen dienen. Schließlich hat es wenig Sinn, jedesmal von neuem das Rad erfinden zu wollen.
Zum zweiten sind aber auch 90% der Maschinenprogramme selbst Erweiterungen oder Ergänzungen zu BASIC-Programmen. Diese Maschinenprogramme übernehmen meist Funktionen, die von BASIC aus entweder gar nicht oder zu langsam erfüllt werden könnten. Um solche Ergänzungen aber schreiben zu können, sollte man schon einen Begriff davon haben, wie der Interpreter arbeitet, dem man auf die Sprünge helfen will.
Gerade bei der Arbeit mit BASIC-Routinen ist ein ROM-Listing sehr hilfreich, allein schon wegen des Umfangs des Interpreters. Wer ein solches Listing nicht hat, kann sich aber auch behelfen, indem er nachschaut, wie entsprechende BASIC-Befehle arbeiten. Die Adressen aller BASIC-Befehle sind zu diesem Zweck im Anhang aufgeführt.
Der Umfang des Interpreters führt auch dazu, daß sich dieses Buch auf einige wenige, aber wichtige Aspekte des Interpreters konzentrieren muß. Es wäre mit Sicherheit ein leichtes, ein eigenes, 500 Seiten starkes Buch über das BASIC 7.0 aus der Sicht des Maschinenprogrammierers zu schreiben.

Zunächst einmal zur Frage, wie ein Interpreterprogramm eigentlich arbeitet:

5.1 Arbeitsweise des Interpreters/ die Interpreter-Hauptschleife

Die Arbeitsweise eines Interpreters unterscheidet sich grundsätzlich von der eines Compilers oder Assemblers. Während ein Compiler oder Assembler aus einem geschriebenen Programmtext in einem Übersetzungsvorgang ein Maschinenprogramm erzeugt, das die im Text beschriebenen Funktionen ausführt, liest der Interpreter ständig im Programmtext, interpretiert einzelne Teile des Textes als Befehle und verzweigt je nach Befehl in verschiedene Unterprogramme, die den jeweiligen Befehl ausführen.

Ein Compiler erzeugt also ein selbständig lauffähiges Maschinenprogramm. Bei einem Interpreter müssen sowohl der Programmtext als auch das interpretierende Maschinenprogramm, der Interpreter, im Speicher resident sein.

Das Lesen des Programmtextes durch den Interpreter erfolgt innerhalb der sogenannten Interpreter-Hauptschleife

LOOP = $4aa2 Interpreter-Hauptschleife

Diese Schleife liest den jeweils nächsten Befehl des Programmtextes, schlägt die Adresse des Befehls in einer Tabelle nach und verzweigt zur Routine, die den Befehl bearbeitet. Steht zum Beispiel ein

print »abcd«

im Text, so liest die Hauptschleife das Befehlswort print, holt die Adresse der PRINT-Unterroutine und springt in diese. Nachdem PRINT abgelaufen ist, führt das rts der PRINT-Routine wieder zur Hauptschleife zurück. Die hinter dem Print-Befehl im Text stehenden weiteren Parameter werden also nicht innerhalb der Hauptschleife ausgewertet, sondern von der speziellen Print-Routine des Interpreters.

In der Hauptschleife werden zwei unterschiedliche Modi unterschieden, die sicher bekannt sind:

der Direktmodus und
der RUN-Modus

Prinzipiell arbeiten beide Modi eigentlich vollkommen gleich. Sie unterscheiden sich hauptsächlich dadurch, woher die Hauptschleife des Interpreters den zu interpretierenden Text bezieht.

Im RUN-Modus ist dieses klar: Der zu interpretierende Text ist eben das eingegebene BASIC-Programm.
Der Interpreter bearbeitet Zeile für Zeile des Programms, wobei die Nummer der augenblicklich bearbeiteten Zeile in

CURLIN = $3b/$3c

festgehalten wird. Der Interpreter kann angewiesen werden, eine bestimmte Zeile anzuspringen (GOTO) oder einige Zeilen mehrmals hintereinander zu interpretieren (FOR) usw.
Im Direktmodus steht der zu interpretierende Text immer in einem Eingabepuffer:

BUF = $200

Dieser Eingabepuffer wird fortlaufend innerhalb der Eingabe-Warteschleife bei

WAIT = $4dc3 Eingabe-Warteschleife

mit Zeichen gefüllt. Dies entspricht der gewohnten Eingabe bei blinkendem Cursor, die mit Return abgeschlossen wird. Nach Abarbeitung des im Puffer vorhandenen Textes meldet sich der Interpreter dann mit READY beziehungsweise auch einer Fehlermeldung zurück.
Die Unterscheidung zwischen Direktmodus und RUN-Modus wird in einem Flag festgehalten:

RUNMOD = $7f

Ist das Bit 7 des Flag gesetzt, befindet man sich im RUN-Modus. Dies führt zum Beispiel dazu, daß bei auftretenden Fehlern eine Zeilennummer ausgegeben wird etc.

5.2 Der Programmtext, die Token

5.2.1 Token

Wie man sich sicher vorstellen kann, ist ein ständiges Interpretieren des Programmtextes keine besonders schnelle Methode der Programmbearbeitung. Die einzelnen Befehlsworte müssen in der Hauptschleife ja erst einmal erkannt werden, das heißt der Text muß sukzessive mit den in einer Tabelle vorhandenen Befehlsworten verglichen werden, bevor die weitere Arbeit in den einzelnen Befehlsroutinen stattfinden kann.

Hier muß man aber eher sagen, die Befehle müßten in der Hauptschleife durch den Vergleich mit einer Tabelle erkannt werden, denn diese Arbeit kann man listigerweise schon bei der Eingabe einer BASIC-Zeile erledigen.
Hierdurch wird der Interpreter beschleunigt, und bei der Eingabe fällt der zusätzliche Arbeitsaufwand zeitlich kaum auf, da die Befehlserkennung relativ schnell vonstatten geht.
Halten wir also fest: Jede eingegebene Zeile wird schon bei der Eingabe einer Befehlserkennung unterzogen.
Die einzelnen Befehlsworte werden bei dieser Überprüfung in sogenannte TOKEN umgewandelt. Das sind im vorliegenden BASIC-7.0-Interpreter Abkürzungen in einer Länge von maximal zwei Zeichen, wobei das erste Zeichen der Abkürzung immer vom ASCII-Wert her größer als $7f = 127 ist. An dieser Eigenschaft erkennt die Hauptschleife des Interpreters, aber auch andere Routinen, bei der späteren Bearbeitung des Textes, daß es sich um ein TOKEN handelt. Die Tabelle der einzelnen Token finden Sie im Anhang.
Die Routine, die die Umwandlung der Befehlsworte in Token übernimmt, liegt bei

TOKEN = $af8a

Andersherum werden Token beim Listen eines Programms auch wieder in Befehlsworte zurückübersetzt. Diese Routine finden Sie an der Adresse:

LIST = $5123

Eine Besonderheit des BASIC-7.0-Interpreters ist, daß es möglich ist, eigene Befehlsworte und eigene Token, sogenannte Usertoken, einzuführen. Wie dies im einzelnen geschieht, ist in Kapitel 6 unter dem Stichwort Usertoken nachzulesen.

5.2.2 Der Programmtext

Wird im Direktmodus eine BASIC-Zeile eingegeben, die mit einer Ziffernfolge beginnt, wird diese Zeile bekanntlich als Programmzeile aufgefaßt und in den schon vorhandenen Programmtext eingefügt.
Jede einzelne Programmzeile hat innerhalb des Programmtextes den folgenden Aufbau:
Die ersten beiden Bytes der Programmzeile stellen den sogenannten Linker der Programmzeile dar. Sie weisen auf die Adresse des ersten Zeichens der nächsten Programmzeile, also auf das Lowbyte des Linkers der nächsten Zeile. Diese Konstruktion wurde deshalb gewählt, um den Ablauf des Listens und der Suche nach Zeilen, zum Beispiel beim GOTO, zu beschleunigen. Um den Programmtext insgesamt mit passenden Link-Adressen zu versehen, gibt es eine spezielle Routine:

LINK = $af87

Nach jeder Veränderung des Programmtextes, beispielsweise wenn eine neue Zeile eingefügt wurde, werden die Programmzeilen durch diese Routine neu zusammengebunden. Auch nach einem LOAD wird diese Routine aufgerufen, damit gewährleistet ist, daß die Linkadressen stimmen.

Die Bytes Nummer 3 und 4 einer Programmzeile sind dann die Zeilennummer. Die Reihenfolge ist dabei wie fast immer: Lowbyte der Nummer in Byte 3, das Highbyte in Byte 4. Zeilennummern können maximal eine Größe von bis zu 63999 haben. Zeilen mit größerer Nummer können nicht direkt eingegeben werden. Ein Grund hierfür ist, daß eine Zeilennummer mit dem Highbyte $ff als zusätzliches Anzeichen für den Direktmodus bei der Ausgabe von Meldungen angesehen wird.

Hinter Linker und Zeilennummer folgt der eigentliche Text der Programmzeile, wie er im Direktmodus auch im Eingabepuffer stehen würde.

Die Zeile wird abgeschlossen durch ein Nullbyte.

Der Start des Gesamttextes wird in der Zeropage in den Adressen

TXTTAB = $2d/$2e

aufbewahrt. Das Ende des Textes ist in der erweiterten Zeropage bei

TXTTOP = $1210/$1211

festgehalten. Das Textende ist außerdem in den Linkern festzustellen: Ein Linker, dessen Highbyte gleich Null ist, weist darauf hin, daß keine Zeile mehr folgt. Der Gesamttext ist folglich durch 2 Nullbyte plus dem Nullbyte des letzten Zeilenendes abgeschlossen. TXTTOP weist hinter das dritte der Nullbytes.

Der gesamte Programmtext liegt immer in der RAM-Bank 0. Seine maximalen Grenzen sind in TXTTAB und

MAXMEM0 = $1212/$1213

definiert. Normalerweise ist die obere Grenze für den Programmtext $ff00. Der Programmstart liegt normalerweise bei $1c00, bei eingeschaltetem Grafikschirm um 9 KByte höher.

5.3 Arithmetik des Interpreters

Einer der Hauptgründe, aus denen auf Routinen des Interpreters in eigenen Maschinenprogrammen zurückgegriffen wird, ist wohl, daß sich im Interpreter unter anderem eine komplette und sehr bequem einzusetzende Fließkomma-Arithmetik befindet. Mit dieser können wir auf einfachste Weise auch komplizierte Rechnungen von der Maschinensprache-Ebene aus durchführen.

5.3.1 Darstellung der Fließkommazahlen durch den Interpreter

Eine Fließkommazahl ist eine Zahl, die als Produkt aus einer Mantisse kleiner gleich 1 und einer Potenz der Zahlenbasis, hier der Basis 2, dargestellt wird. Wer schon einmal mit Logarithmen zu tun hatte, wird dies sicher kennen.

Beispiele mit der Zahlenbasis 10 sind sicher geläufig:

```
        350       = 0.35 * 10↑3
oder
        1234      = 0.1234 * 10↑4
```

Im Dualsystem ist es nicht anders:

```
        16.25     = %1 0000.01
                  = %0.1000 001 * 2↑%0101
```

Nachkommastellen haben im Dualsystem im übrigen eine analoge Bedeutung wie Nachkommastellen des Dezimalsystems:

Dezimal:

```
        0.1       = 1/10     = 1/Basis des Dezimalsystems
        0.01      = 1/100    = 1/B10/B10
        0.001     = 1/1000   = 1/B10/B10/B10 usw.
```

Binär:

```
        0.1       = 1/2      = 1/Basis Dualsystem
        0.01      = 1/4      = 1/B2/B2
        0.001     = 1/8      = 1/B2/B2/B2 usw.
```

Eine Zahl im Fließkommaformat ist nun einfach durch ihre Mantisse und ihren Exponenten anzugeben, wobei natürlich die Vorzeichen hinzukommen.

Die Genauigkeit der Zahlendarstellung hängt ausschließlich davon ab, wie genau die Mantisse dargestellt wird. Der darstellbare Zahlenbereich hängt von der maximalen Größe des Exponenten ab.

Die Fließkomma-Arithmetik unseres BASIC-Interpreters benutzt für die Darstellung der Mantisse insgesamt 4 Byte, für den Exponenten 1 Byte. Damit liegt der darstellbare Zahlenbereich zwischen

1.7 E+38 und 2.9 E–39

beziehungsweise

–2.9 E–39 und –1.7 E+38

Durch die Größe der dargestellten Mantisse kann etwa mit zehn Ziffern gerechnet werden. Dies bedeutet aber nicht automatisch, daß die Rechnung auch bis auf die zehnte Ziffer genau ist.

Im Zusammenhang mit den Vorzeichen der Fließkommazahlen sind zwei Fälle zu unterscheiden.
Erstens kann die Fließkommazahl sozusagen rechenbereit in einem Register in der Zeropage stehen. Auf diese Register kommen wir anschließend noch.

In diesem Fall erhält das Vorzeichen der Mantisse ein gesondertes Byte. Das höchstwertige Bit dieses Vorzeichenbytes ist gesetzt, wenn eine negative Mantisse und damit eine negative Fließkommazahl vorliegt. Ist die Zahl positiv, so ist dieses Bit 7 gleich Null. Steht eine Fließkommazahl also rechenbereit in der Zeropage, so belegt sie 6 Byte: 4 Byte Mantisse, 1 Byte Vorzeichen der Mantisse und 1 Byte Exponent.

Der zweite Fall ist, daß eine Fließkommazahl abgespeichert wird, beispielsweise in einer Variablen. Es wäre ja eine reine Verschwendung von Speicherplatz, ein ganzes Byte für das Vorzeichen der Mantisse zu opfern, wenn es auch anders geht. Um diese andere Möglichkeit zu verstehen, muß man wissen, daß die Mantissen bei der Fließkommarechnung des Interpreters immer linksbündig in den Registern stehen. Dies heißt nichts anderes, als daß die Mantissen so lange nach links verschoben werden und dabei der Exponent dekrementiert wird, bis die höchstwertige Stelle der Mantisse eine 1 ist. Dies ist schon deshalb notwendig, um mit möglichst vielen gültigen Ziffern rechnen zu können. Ein Beispiel (nur mit 1 Byte statt mit 4):

Statt einer Mantisse von

% 0010 1010, Exponent %0111

erhält man durch schrittweises Linksverschieben der Mantisse (= Multiplikation mit 2) und Erniedrigen des Exponenten:

% 0101 0100, Exponent %0110
% 1010 1000, Exponent %0101

Diese sogenannte normalisierte Darstellung der Fließkommazahl zeichnet sich dadurch aus, daß insbesondere das höchstwertige Bit der Mantisse gleich 1 ist. Da dies immer der Fall ist, kann man es bei der Speicherung der Fließkommazahlen als bekannt voraussetzen und statt dessen in diesem Bit das Vorzeichen der Mantisse unterbringen.
Die Darstellung des Exponenten ist in beiden Fällen gleich. In einem Byte können Zahlen von 0 bis 255 dargestellt werden, oder, unter Benutzung des 2er-Komplements für negative Zahlen, Zahlen von +127 bis –128. Die Darstellung des Exponenten der Fließkommazahlen erfolgt nicht direkt im 2er-Komplement, es wird statt dessen der Betrag des kleinstmöglichen Exponenten zum tatsächlichen Exponenten addiert.
Man erhält so zum Beispiel für einen Exponenten –3 die Darstellung 128–3 = 125 oder für +27 die Darstellung 128+27 = 155. Beachten Sie bitte, daß es sich hierbei nicht um Exponenten der Basis 10, sondern um die Basis 2 handelt.
Ein Exponent Null gilt als Flag dafür, daß eine Fließkommazahl gleich Null ist.
Die Umwandlung der Fließkommazahlen per pedes von einem in das andere Zahlensystem in Quell- und Zielsystemen möchte ich mir sparen, da dies für die meisten sicher nicht von Interesse ist. Statt dessen folgt bei den späteren Beispielen ein kleines Maschinenprogramm, das uns Umrechnungen ermöglicht, so daß wir eigene Fließkommazahlen in unseren Programmen verwenden können.

5.3.2 Register der Fließkomma-Arithmetik (FAC, ARG, RES)

Alle möglichen Fließkomma-Operationen werden über Register in der Zeropage durchgeführt. Die beiden wichtigsten dieser Register sind

	FAC	Fließkomma-Akkumulator 1. Exponent = \$63; Mantisse = \$64 bis \$67; Vorzeichen Mantisse = \$68
und	**ARG**	Fließkomma-Akkumulator 2. Exponent = \$6a; Mantisse = \$6b bis \$6e; Vorzeichen Mantisse = \$6f

Hinzu kommt

ARISGN = $70

in dem ein Vorzeichenvergleich der beiden Akkumulatoren aufbewahrt wird. Ist ARISGN = $ff, so sind die Vorzeichen ungleich, ist ARISGN = 0, so sind die Vorzeichen gleich.

FACOV = $71

ist ein Rundungsbyte zu FAC. Bei der Multiplikation und Division ist außerdem ein temporäres Register notwendig. Dies liegt bei

RES = $28 bis $2c

Bei der Berechnung von Sinus, Cosinus, Logarithmus usw. werden zwei weitere temporäre Hilfsregister herangezogen. Diese liegen bei:

TEMPF1 = $59 bis $5d

und

TEMPF2 = $5e bis $62

Als Zeiger auf das jeweilige temporäre Register wird die Zeropage-Adresse

INDEX = $24/ $25

benutzt.

Mehr brauchen wir für die praktische Arbeit mit der Fließkomma-Arithmetik nicht von den Registern zu wissen.

5.3.3 Integerzahlen

Es fehlt jetzt noch eine Beschreibung, wie mit Integerzahlen im Interpreter gerechnet wird. »Nur auf Umwegen«, könnte man als Antwort geben.

Jede Integerzahl wird nämlich zur Rechnung zunächst in eine normalisierte Fließkommazahl umgewandelt. Nach erfolgter Rechnung erfolgt eine Wandlung zurück ins Integerformat, falls erforderlich.

Der Interpreter verfügt also nicht über eine eigene schnelle Integer-Arithmetik, das ist wichtig zu wissen. Eine besondere Darstellung der Integerzahlen ist nur für den Gebrauch von Variablen vorgesehen, da diese Darstellung weniger Speicherplatz benötigt als die Darstellung einer Fließkommazahl.

Integerzahlen werden wie gewöhnlich in 2 Byte aufbewahrt. Negative Zahlen werden durch das 2er-Komplement beschrieben. Dadurch können Integerzahlen einen Wertebereich von +32767 bis -32768 abdecken.

Die Umwandlung einer Integerzahl in eine normalisierte Fließkommazahl kann man auch leicht von Hand vornehmen:

% 0010 0010 0000 0000 sei die Integerzahl

Diese Integerzahl kann man auch schreiben als

% 0010 0010 0000 0000 E % 0000

Durch Rechtsverschieben der Integerzahl bei gleichzeitigem Inkrementieren des Exponenten (zur Basis 2!) erhält man:

```
% 0001 0001 0000 0000   E % 0001
% 0000 1000 1000 0000   E % 0010
%      0100 0100 0000   E % 0011
%      0010 0010 0000   E % 0100
%      0001 0001 0000   E % 0101
%           1000 1000   E % 0110
%           0100 0100   E % 0111
%           0010 0010   E % 1000
%           0001 0001   E % 1001
% 0000 1000. 1          E % 1010
%      0100. 01         E % 1011
%      0010. 001        E % 1100
%      0001. 0001       E % 1101
%         0. 1000 1     E % 1110
```

Rückwärts verläuft die Konvertierung einer Fließkommazahl in eine Integerzahl analog ab. In beiden Fällen müssen aber negative Zahlen gesondert betrachtet werden, da diese bei der Integer-Darstellung durch das 2er-Komplement und nicht durch ein besonderes Vorzeichenbit kenntlich gemacht sind.

5.3.4 Einige Routinen der Fließkomma-Arithmetik

Dies ist für den Anwender sicherlich der interessanteste Teil der Beschreibung der Arithmetik des Interpreters. Fast alle der hier angesprochenen Routinen sind in einer Sprungtabelle zusammengefaßt, die ab der Adresse $af00 im oberen Teil des BASIC-ROMs beginnt.

FKTOINT = $af00

Umwandlung Fließkomma nach Integer

IN: FAC = Fließkommazahl

Diese Routine wandelt eine Fließkommazahl in die entsprechende Integerzahl um. Ein Betrag von FAC größer als 32768 führt zur Fehlermeldung »illegal quantity« des Interpreters. Man beachte, daß eine Integerzahl größer gleich 32768 (Betrag) als 2er-Komplement einer negativen Zahl betrachtet wird. Aus diesem Grund lieferte zum Beispiel die FRE-Funktion von BASIC 2.0 auch negative Werte!

OUT: $66 = Lowbyte der Integerzahl
$67 = Highbyte

COM: Das Ergebnis der Wandlung steht rechtsbündig in der Mantisse des FAC.

INTTOFK = $af03

Umwandlung Integer nach Fließkomma

IN: A = Highbyte (!) der Integerzahl
Y = Lowbyte

Die in Akkumulator und Y-Register übergebene Integerzahl wird linksbündig in die FAC-Mantisse gebracht. Anschließend wird die Umwandlung durch die schon oben angesprochene Verschiebung der Mantisse bei Erhöhung des Exponenten vorgenommen. Zahlen größer als $8000 führen zur Bildung einer negativen Fließkommazahl. Für diese Integerzahlen ist im Bedarfsfall die weiter unten beschriebene Routine NORMINT zu nehmen.

OUT: FAC = Fließkommaergebnis

FKTOSTR = $af06

Umwandlung einer Fließkommazahl in FAC in einen ASCII-String

IN: FAC = Fließkommazahl

Der Ausgabe-String beginnt bei FBUFFR = $100 in der erweiterten Zeropage. Dieser Bereich liegt eigentlich noch im Prozessorstack, es kann aber davon ausgegangen werden, daß die Inanspruchnahme des Stacks bei BASIC nie so hoch ist, daß diese Speicherzellen durch den Stack benötigt werden. Die Adresse $100 enthält das ASCII-Zeichen des Vorzeichens von FAC, im positiven Fall einfach ein Leerzeichen, im negativen Fall ein Minuszeichen. Dahinter, ab $101, folgt die in ASCII-Zeichen umgewandelte Fließkommazahl mit Exponenten, falls erforderlich. Der ASCII-String ist durch ein Nullbyte abgeschlossen.

OUT: A = Lowbyte des Ausgabepuffers
Y = Highbyte des Puffers

COM: Zur Ausgabe von Integerzahlen existiert eine viel bequemere Routine, die gleich anschließend beschrieben wird. Man beachte, daß die vorliegende Routine nur den Ausgabe-String bereitstellt, ihn aber nicht selbst ausgibt.

INTOUT = $8e32

IN: A = Highbyte der Integerzahl
X = Lowbyte

Akku und X-Register werden linksbündig in die FAC-Mantisse geschrieben, der Rest des FAC wird mit Nullen gefüllt. Der Exponent wird auf $90 = %1001 0000 gesetzt, dies entspricht einer schon stattgefundenen Verschiebung um 16 Stellen nach rechts. Anschließend erfolgt die Umwandlung des FAC in einen String und die Ausgabe über die BSOUT-Routine. Das Normalisieren der Integerzahl in FAC ist genauer in der Routine $af0f nachzulesen.

OUT: –

COM: Durch diese Routine erfolgt keine Ausgabe negativer Zahlen. Integerzahlen größer als 32767 werden auch als positive Zahlen ausgegeben. Die Routine dient dem Interpreter beispielsweise zur Ausgabe von Zeilennummern.

STRTOFK = $af09

Wandlung eines ASCII-Strings in Bank 1 in eine Fließkommazahl, wie es zum Beispiel durch die VAL-Funktion vorgenommen wird.

IN: A = String-Länge
$24/$25 = String-Adresse in Bank 1

Analog zur Wandlung einer Fließkommazahl in einen String kann auch ein String wieder zurück in eine Fließkommazahl gewandelt werden. Die vorliegende Routine behandelt den Fall, daß dieser String im String-Bereich in Bank 1 aufbewahrt ist, was bei BASIC immer der Fall ist.

OUT: –

FCTOADR = $af0c

IN: FAC = Fließkommazahl

FCTOADR wandelt die in FAC übergebene Fließkommazahl in eine Integerzahl um. Ergibt sich, daß die Integerzahl größer als +32678, also negativ war, wird ein »illegal quantity« ausgegeben.

OUT: $22 = Lowbyte der Integerzahl
$23 = Highbyte der Zahl

NORMINT = $af0f

IN: clc, sec
X = Exponent
$64 = Highbyte der Integerzahl
$65 = Lowbyte der Zahl

Diese Routine bringt eine in den beiden höchstwertigen Bytes der FAC-Mantisse übergebene Integerzahl in die normalisierte Fließkommadarstellung. Die beiden nicht benutzten Bytes der FAC-Mantisse werden durch NORMINT gelöscht. Im X-Register wird der zugehörige Exponent übergeben. Im Fall, daß eine normale 16-Bit-Zahl in eine Fließkommazahl umgewandelt werden soll, beträgt dieser Exponent $90 = %1001 0000 = 128 + 16. Wir erinnern uns, daß bei der Darstellung des Exponenten stets der Betrag des kleinstmöglichen Exponenten, 128, addiert wird. Der Exponent sagt praktisch aus, daß die in FAC-Mantisse übergebene Zahl durch im Beispiel 16maliges Linksschieben der Mantisse die Integerzahl ergibt. Vor dem höchstwertigen Mantissen-Byte muß man sich ja ein Komma vorstellen, alle Bits in der Mantisse sind ja Nachkomma-Bits. Natürlich kann man auch andere Exponenten angeben. Der Wert der normalisierten Zahl nach Aufruf der Routine ist dann größer oder kleiner.

Im Carry-Flag kann man die Anweisung geben, ob die übergebene Zahl vor der Normalisierung in das 2er-Komplement gewandelt werden soll oder nicht. Gleichzeitig mit der Bildung des 2er-Komplementes wird auch das Vorzeichen von FAC umgedreht.

Ist das Carry gesetzt, wird die Wandlung in das Komplement unterdrückt, die übergebene Integerzahl wird also auch dann als positiv angesehen, wenn die Zahl größer als 32767 ist.

Ist das Carry clear, so wird die Integerzahl unbedingt erst in das 2er-Komplement gewandelt und das Vorzeichen von FAC invertiert.

OUT: FAC = normalisierte Fließkommazahl

SUBVAR = $af12

Subtraktion FAC := ((A,Y)) – FAC

IN: A = Lowbyte der Adresse einer FK-Zahl in Bank 1
Y = Highbyte der Adresse

Die Routine überträgt zunächst eine Fließkommazahl aus der mit PCRB (Prä-Konfigurationsregister A) anwählbaren Konfiguration nach ARG. Normalerweise wird mit PCRB die RAM-Bank 1 ohne ROMs eingestellt. Daran sollte man bei der Verwendung von BASIC-Routinen auch nichts ändern.

Die Adresse der nach ARG zu holenden Zahl wird in Akku und Y-Register übergeben. Die Fließkommazahl selbst muß in der Speicherdarstellung gegeben sein, das heißt das Vorzeichen der Mantisse ist im Bit 7 des höchstwertigen Mantissen-Bytes zu finden.

Die einzelnen Bytes liegen im Speicher in derselben Reihenfolge hintereinander wie in den Registern in der Zeropage:

1. Byte Exponent
2. Byte höchstwertiges Byte der Mantisse
3. Byte nächsttieferes Byte der Mantisse
4. Byte
5. Byte niedrigstwertiges Mantissen-Byte

Nach der Übertragung erfolgt die Subtraktion FAC := FAC - ARG.
OUT: FAC = Differenz

SUBARG = $af15

Subtraktion FAC := ARG - FAC
IN: ARG, FAC
Dies ist die eben angesprochene Subtraktion der Register ARG und FAC. Die Subtraktion kann natürlich erst nach einer Anpassung der beiden Exponenten erfolgen. Diese Anpassung der Exponenten wird aber von der Routine selbst vorgenommen, braucht uns also nicht zu kümmern. Das Ergebnis der Subtraktion steht wieder in FAC.
OUT: FAC = Differenz

ADDVAR = $af18

Addition FAC := ((A,Y)) + FAC
IN: A = Lowbyte der Adresse einer FK-Zahl in RAM-Bank 1
Y = Highbyte der Adresse
Diese Routine entspricht in allen Einzelheiten genau der Routine SUBVAR, nur werden FAC und die Fließkommazahl aus dem RAM addiert statt subtrahiert.
OUT: FAC = Summe

ADDARG = $af1b

Addition FAC := ARG + FAC
IN: ARG, FAC
siehe SUBARG mit der Addition statt der Subtraktion.
OUT: FAC = Summe

MULVAR = $af1e

Multiplikation FAC := ((A,Y)) * FAC
IN: A = Lowbyte der Adresse
Y = Highbyte der Adresse
Die betreffende Fließkommazahl wird wieder aus der RAM-Bank 1 in den ARG geholt und anschließend mit FAC multipliziert. Dazu werden die Exponenten addiert und die Mantissen bitweise miteinander multipliziert. Hierzu wird das Hilfsregister RES herangezogen.
OUT: FAC = Produkt

MULARG = $af21

Multiplikation FAC := ARG * FAC
IN: ARG, FAC
Dasselbe wie in der vorigen Routine, nur wird in ARG schon eine Zahl erwartet. Die Routine MULARG prüft zu Beginn der Routine das EQUAL-Flag, das im Prozessorstatus übergeben wird. Die Multiplikation wird nicht durchgeführt, wenn das EQUAL-Flag gesetzt ist! Dies ist durchaus sinnvoll, wenn die vor MULARG aufgerufene Routine den FAC-Exponenten als Null-Flag in den Akku geladen hat, da dadurch eine Multiplikation mit Null nicht durchgeführt wird. Um eine unbedingte Multiplikation zu erreichen, ist die Verwendung von

MULARG = $8a2c

zu empfehlen.
OUT: FAC = Produkt

DIVVAR = $af24

Division FAC := ((A,Y)) / FAC
IN: A = Lowbyte der Adresse
Y = Highbyte
OUT: FAC = Quotient

DIVARG = $af27

Division FAC := ARG / FAC
IN: ARG, FAC
OUT: FAC = Quotient

POTCON = $af36

Potenzierung FAC := ARG ↑ ((A,Y))
IN: A = Lowbyte der Adresse
Y = Highbyte

Man beachte an dieser Stelle, daß als Name der Routine nicht POTVAR gewählt wurde. Warum: Der Exponent, auf den der Akkumulator und das Y-Register weisen, wird nicht über die Verwendung des PCRB, sondern direkt ohne Veränderung der Konfiguration nach FAC geholt. Dies bedeutet im Normalfall, daß die Konfiguration eingeschaltet ist, in der alle ROMs und die RAM-Bank 0 ausgewählt sind. Normalerweise wird sich der Exponent also in einem der ROMs befinden, weswegen der Name auch zu POTCON geriet.
OUT: FAC = Potenz

POTFAC = $af39

Potenzierung FAC := ARG ↑ FAC

IN: ARG, FAC

Dies ist die eigentliche Potenzier-Routine. Die Potenz wird unter Zuhilfenahme der LN- und EXP-Funktion vorgenommen, ist also nicht die genaueste, was übrigens auch für die Wurzelfunktion gilt, die einfach als Potenz mit Exponenten 0.5 berechnet wird.

OUT: FAC = Potenz

CHSSGN = $af33

Vorzeichenwechsel FAC := – FAC

IN: –

Zum Vorzeichenwechsel wird einfach das Bit 7 des Mantissen-Vorzeichenbytes umgedreht. Die restlichen Bits, die man in diesem Byte antreffen kann, sind ohne Bedeutung.

OUT: –

ROUNDF = $af4b

FAC runden FAC := FAC + FACOV

IN: FAC, FACOV = $71

Diese Routine rundet FAC in Abhängigkeit des Wertes des Rundungsbytes FACOV. Ist FACOV kleiner als $80, bleibt FAC unverändert, ansonsten wird die Mantisse von FAC inkrementiert und normalisiert. Da praktisch bei jeder Operation eine Rundungsstelle auftritt, empfiehlt es sich, zugunsten der Genauigkeit vor jeder Bewegung von FAC in andere Register oder den Speicher oder auch vor der Ausgabe, den FAC zu runden.

OUT: FAC gerundet

SGNFAC = $af51

Vorzeichen von FAC ermitteln

IN: FAC

Die Routine SGNFAC ermittelt das Vorzeichen von FAC aus dem Vorzeichenbyte der Mantisse und dem Exponenten (Null-Flag). Das Ergebnis erscheint im Akkumulator und im Prozessorstatus:

FAC = 0	EQUAL-Flag gesetzt
FAC negativ	CARRY gesetzt, A = $ff
FAC positiv	CARRY clear, A = 1

OUT: clc, sec, equ, neq
A = $ff, 1

CMPCON = $af54

Zahlenvergleich FAC mit ((A,Y))

IN: FAC
A = Lowbyte der Adresse der zweiten Zahl
Y = Highbyte

Wie schon bei POTCON weist der Zeiger (A,Y) auf eine Adresse, die in der Normal-Konfiguration zu erreichen ist. Insbesondere kann diese Adresse auch in der Zeropage liegen (ARG) oder im ROM. Das Ergebnis des Vergleichs ist wie in der vorigen Routine im Status und im Akku abzulesen:

FAC = ((A,Y))	EQUAL-Flag gesetzt
FAC kleiner	CARRY set, A = $ff
FAC größer	CARRY clear, A = 1

OUT: sec, clc, equ, neq
A = $ff, A = 1

Es folgen einige Routinen, die mit Hilfe von Polynomberechnungen den Wert einiger Funktionen bestimmen. Zur Berechnung der Polynome werden die Hilfsregister TEMPF1 und TEMPF2 herangezogen.

LOGFAC	= **$af2a**	Logarithmus FAC := ln(FAC)
SQRFAC	= **$af30**	Quadratwurzel FAC := FAC ↑ 0.5
EXPFAC	= **$af3c**	Exponentialfunktion FAC := exp(FAC)
COSFAC	= **$af3f**	Cosinus FAC := cos(FAC)
SINFAC	= **$af42**	Sinus FAC := sin(FAC)
TANFAC	= **$af45**	Tangens FAC := tan(FAC)
ATGFAC	= **$af48**	Arcustangens FAC := arctan(FAC)

Hinzu kommen einige Routinen, die ebenfalls BASIC-Funktionen berechnen, bei denen aber keine Polynomauswertung notwendig ist:

INTFAC	= **$af2d**	Eliminierung der Nachkommastellen FAC := int(FAC)
ABSFAC	= **$af4e**	Absolutwert FAC := abs(FAC)

Auch Zufallszahlen können wir erzeugen:

RNDFAC = $af57

Zufallszahlen FAC := rnd(FAC)

IN: FAC

Es sind drei Typen von Zufallszahlen zu unterscheiden, deren Erzeugung vom Argument abhängt, das in FAC übergeben wird:

FAC = 0	Die Bytes der Mantisse von FAC werden mit den gerade in den Timern A und B des CIA 1 enthaltenen Zählwerten gefüllt.
FAC größer 0	Die Zufallszahl wird aus der letzten erzeugten Zufallszahl durch Multiplikation mit einer Konstanten und Addition einer Konstanten bei anschließender Vertauschung der

	Mantissen-Bytes gewonnen. Die letzte Zufallszahl wird zu diesem Zweck in der Adresse $121b und folgenden aufbewahrt.
FAC kleiner 0	Die Mantissen-Bytes werden einfach vertauscht.

In allen Fällen ist der Exponent gleich 128 (also Null), das Vorzeichen ist Plus.

OUT: FAC = Zufallszahl

Zum Abschluß folgen noch einige Routinen, die den Transport von Fließkommazahlen zwischen den Registern und vom oder zum Speicher hin erlauben:

MOVVARA = $af5a

ARG: = ((A,Y)) aus RAM-Bank 1
IN: A = Lowbyte des Zeigers auf die Fließkommazahl in Bank 1
Y = Highbyte des Zeigers

Unter Verwendung der indirekten Laderoutine I24SR1 wird eine Fließkommazahl aus Bank 1 in den ARG geholt. Im Normalfall wird dies eine in einer Variablen gespeicherte Zahl sein.

OUT: A = Null-Flag. Der Akkumulator wird mit dem Exponenten der geholten Zahl geladen, wobei ein Exponent Null als Kennzeichen für eine Fließkommazahl gleich 0 dient.

MOVCONA = $af5d

ARG: = ((A,Y)) ohne Änderung der Konfiguration
IN: (A,Y) als Zeiger

Diese Routine leistet dasselbe wie die vorangegangene, jedoch wird das Konfigurationsregister nicht umgeschaltet. Im Normalfall werden so Konstante aus dem BASIC-ROM in den ARG transportiert. Möglich wäre aber auch der Einsatz von MOVCONA, um eines der Hilfsregister in der Zeropage nach ARG zu bringen.

OUT: A = Null-Flag

MOVVARF = $af60

FAC: = ((A,Y)) aus Bank 1
IN: (A,Y) als Zeiger

Genau wie MOVVARA legt die vorliegende Routine eine Fließkommazahl aus Bank 1 in den FAC.

OUT: –

MOVCONF = $af63

FAC: = ((A,Y))
IN: (A,Y) als Zeiger

Diese Routine entspricht MOVCONA.
OUT: Y = 0
A = Null-Flag

STOFAC = $af66

FAC nach ((X,Y)) bringen
IN: X = Lowbyte des Zeigers auf die Bestimmungsadresse
Y = Highbyte
Diese Routine rundet zunächst den FAC, bevor er in eine Adresse in der RAM-Bank 0 gebracht wird. Das Konfigurationsregister wird bei der Abspeicherung nicht verändert. Die Umwandlung der Fließkommazahl vom Rechen- in das Speicherformat wird von STOFAC übernommen.
OUT: Y = 0
FACOV = 0

ARGTOF = $af69

FAC: = ARG

FACTOA = $af6c

ARG: = FAC
Vor der Übertragung des FAC nach ARG wird zunächst FAC gerundet.

5.4 Strings

Das nächste interessante Kapitel nach der Betrachtung der Fließkomma-Arithmetik ist die Behandlung der Strings durch den Interpreter.
Was Strings sind, dürfte klar sein: Beim C64 konnte man noch von Zeichenketten sprechen, da Strings nur aus Zeichen bestehen konnten, beim C128 spricht man besser von Byte-Ketten, wenn man zum Beispiel an die Shapes denkt.
Jede Byte-Kette kann beschrieben werden durch die Länge der Kette, die Adresse des ersten Bytes der Kette und durch die Konfiguration, unter der die Byte-Kette angetroffen werden kann.
Im Commodore 128 spart man sich die letzte Angabe dadurch, daß alle Strings in einen einheitlichen String-Bereich in RAM-Bank 1 gebracht werden, bevor man sie bearbeitet. Diese Methode hat auch noch andere Vorzüge, wenn man beispielsweise an Operationen mit Strings oder an die Garbage Collection denkt (siehe unten).

5.4.1 Der String-Bereich in Bank 1

Der für alle Strings gemeinsame String-Bereich in der RAM-Bank 1 wird durch zwei Zeiger in der Zeropage beschrieben:

MAXMEM1 = $39/ $3a zeigt hinter die höchste Adresse, die in der Bank 1 für Strings zur Verfügung steht. Normalwert ist $ff00, die String-Speicherung geht also bis $feff in Bank 1.

FRETOP = $35/ $36 ist das untere Ende des String-Bereiches. Dieses wechselt je nachdem, wie voll oder leer der Bereich ist. FRETOP weist unter das letzte Byte in diesem Bereich.

Wenn Sie sich fragen, warum das untere Ende des String-Bereiches »TOP«, also Spitze genannt wird, sind Sie schon auf etwas gestoßen:
Die Strings werden nämlich von oben ($feff) nach unten in den String-Bereich eingebracht. Die Strings in sich sind in der richtigen Byte-Reihenfolge von unten nach oben, die Einträge in den String-Bereich wachsen allerdings von oben nach unten. Der erste Eintrag endet bei $feff, der zweite kommt davor etc.
Da alle Strings – einerlei, ob es sich um den String einer String-Variablen oder um einen String in Anführungszeichen oder um ein CHR$ handelt – im gleichen String-Bereich untergebracht werden, ist es notwendig, diese unterschiedlichen String-Typen kenntlich zu machen.

Byte-Ketten, deren Descriptor – Länge und Adresse der Kette – in einer Variablen festgehalten ist, erhalten als Zusatz im String-Bereich 2 Byte. Diese werden am Ende des eigentlichen Strings angefügt und enthalten die Adresse, an der der String-Descriptor steht. Also die Adresse der zugehörigen Variablen. Der Gesamteintrag in den String-Bereich sieht damit so aus:

Byte-Kette/ Lowbyte der Variablenadresse/ Highbyte

Dazu muß man noch wissen, daß auch sämtliche Variable in Bank 1 untergebracht sind. Die Angabe einer Konfiguration erübrigt sich also.
Der zweite String-Typ wird oft auch als temporäre Strings bezeichnet. Unter temporären Strings verstehen wir ganz einfach alle Strings, deren Descriptoren nicht in einer Variablen festgehalten sind. Solche Strings erhalten im String-Bereich ebenfalls einen Zusatz von 2 Byte, an denen sie erkannt werden können:

Byte-Kette/ Länge der Kette/ $ff

Beispiele für Strings, deren Descriptoren nicht in String-Variablen festgehalten sind:

```
print "commodore"
chr$(9)
mid$(a$)
a$ = "string": der zweite String
```

5.4.2 Der Stack für temporäre String-Descriptoren

Dieser Stack, der häufig auch einfach temporärer Stringstack genannt wird, entsteht aus der Notwendigkeit, daß an irgendeiner Stelle auch die Descriptoren temporärer Strings aufbewahrt werden müssen. Irgendwo muß der Computer sich eben merken, wie lang ein String und wo er zu finden ist. Dies geschieht entweder in einer String-Variablen oder eben im temporären Stringstack.
Der temporäre Stringstack liegt in der Zeropage an der Adresse

TEMPST = $1b bis $23

Er ist 9 Byte lang, das heißt, es können 3 Descriptoren in den Stack aufgenommen werden (1 Byte Länge des Strings, 2 Byte Adresse des Strings in Bank 1).
Warum gleich mehrere Descriptoren aufgenommen werden müssen, wird klar, wenn man zum Beispiel an die String-Verkettung denkt:

```
"abcd" + "efghi"
```

Zwei Zeiger weisen in den temporären Stringstack:

TEMPPT = $18

ist ein 1-Byte-Zeiger, der auf den nächsten freien Platz im Stapel weist. Steht in TEMPPT zum Beispiel eine $1b, so bedeutet dies, daß der erste Platz im Stack frei ist.

LASTPT = $19/ $1a

weist auf den zuletzt in den Stapel eingetragenen Descriptor. Da der Stapel in der Zeropage liegt, ist das Highbyte dieses Pointers immer gleich Null.
Halten wir noch einmal fest, wie der Interpreter bei der Aufnahme eines temporären Strings vorgeht:
Als erstes wird der String in den String-Bereich der Bank 1 kopiert. Der String erhält 2 Byte zusätzlich, die ihn als temporär kenntlich machen, dies ist die String-Länge und ein $ff.

Neue Adresse und die Länge des Strings werden in den Stringstack gebracht, wobei LASTPT auf diesen Eintrag weist.
Mehr brauchen wir über den Stringstack nicht zu wissen, der Stack gehört eigentlich mehr zu den Interna des Interpreters, mit denen man es nicht unmittelbar zu tun bekommt.

5.4.3 Die Garbage Collection

Unter der Garbage Collection (Müllabfuhr) ist folgendes zu verstehen:

Betrachten wir einmal die Befehlsfolge

```
a$ = "eins"
a$ = "zwei"
```

Wir weisen einer Variablen zweimal einen String zu. Wie wir jetzt wissen, führt dies dazu, daß die beiden Strings »eins« und »zwei« im String-Bereich in Bank 1 erscheinen. Beide Einträge der Strings enthalten den Zusatz der Variablenadresse, wo der String-Descriptor abgespeichert ist.

Im String-Bereich stehen also zwei Einträge:

```
'eins', Adresse von a$
```

und

```
'zwei', Adresse von a$
```

Der erste der beiden Einträge ist aber nach der Zuweisung von »zwei« an a$ überflüssig geworden.

Solche überflüssigen Einträge in den String-Bereich treten im Verlauf eines BASIC-Programms recht häufig auf. Sind sie so häufig, daß im String-Bereich kein weiterer Platz mehr ist, schlägt die Müllabfuhr zu.
Dazu müssen die String-Einträge mit den Variableneinträgen verglichen werden. Ungültige Strings werden entfernt und der Rest des String-Bereiches zusammengerückt. Dies bedingt, daß sich die Adressen, sprich Descriptoren, der Strings ändern, diese müssen demnach angepaßt werden. Das Ganze dauert natürlich seine Zeit, ist aber unvermeidlich.
Ist trotz Garbage Collection kein Platz mehr im String-Bereich, gibt es einen Error »out of memory«.

5.5 Variable

Variable werden im 128er immer in der RAM-Bank 1 angelegt. Der für Variable zur Verfügung stehende Platz wird durch einige Zeiger in der Zeropage angegeben:

VARTAB = $2f/$30	untere Grenze des Variablenspeichers. Normalerweise liegt diese Grenze bei $400 in Bank 1.
ARYTAB = $31/$32	untere Grenze der dimensionierten Variablen (Felder). Dies ist zugleich die obere Grenze der nicht dimensionierten Variablen. ARYTAB ändert sich, je nachdem, wieviele nicht dimensionierte Variable vorhanden sind.
STREND = $33/$34	zeigt hinter das Ende der Felder. Dies ist zugleich die unterste Adresse, die noch für den String-Bereich verfügbar ist.

Variable können von vier verschiedenen Typen sein:

Fließkomma (REAL)
Ganzzahl (INTEGER)
String
FN-Variable

Als Variablennamen sind höchstens zwei Buchstaben erlaubt. Der Variablenname wird mit in der Variablen abgespeichert, wobei die beiden Bits 7 der zwei Variablenzeichen als Kennzeichnung für den Variablentyp gesetzt oder gelöscht sind:

1. Zeichen Bit 7	2. Zeichen Bit 7	Variablentyp
0	0	REAL
0	1	String
1	0	FN
1	1	INTEGER

5.5.1 REAL-Variable und -Felder

Fließkomma-Variable benötigen im Variablenraum 7 Byte Speicherplatz. 2 Byte werden zur Speicherung des Namens verwendet, die restlichen 5 zur Speicherung des Fließkommawertes im Speicherformat (Vorzeichen in die Mantisse integriert).

Die einzelnen Bytes einer REAL-Variablen liegen in folgender Reihenfolge im Speicher:

1. Zeichen des Variablennamens
2. Zeichen des Namens
1 Byte Exponent
4 Byte Mantisse mit Vorzeichen

Bei einem REAL-Array wird natürlich nicht bei jedem Feldelement der Feldname mit abgespeichert. Statt dessen trägt das Feld einen sogenannten Feldkopf, in dem der Feldname und die Anzahl der Elemente und Dimensionen des Feldes festgehalten sind. Jedes Feldelement hat eine Länge von 5 Byte.
Der Feldkopf hat das folgende Aussehen:

1. Zeichen des Feldnamens
2. Zeichen des Namens
Gesamtlänge des Feldes in Bytes gerechnet, in der Reihenfolge Lowbyte/Highbyte
Anzahl der Dimensionen (1 Byte)
Anzahl der Elemente je Dimension, von hinten beginnend in der Reihenfolge Highbyte/ Lowbyte
dahinter die Elemente selbst

»Anzahl der Elemente von hinten beginnend« bedeutet hier nichts weiter, als daß die zuletzt deklarierte Dimension zuerst im Feld-Header auftaucht. Bei a$(6,7,8) kommt also als erste Anzahl die 9 vor (Element 0 bis 8 = 9 Stück).
Die Reihenfolge der Feldelemente hinter dem Feldkopf am Beispiel eines zweidimensionalen Feldes mit 4 Elementen (dim a$(1,1):

a$(0,0) / a$(1,0) / a$(0,1) / a$(1,1)

5.5.2 INTEGER-Variable und -Felder

Die Darstellung der Integer-Variablen erfolgt im nicht dimensionierten Fall ebenfalls in 7 Byte, wovon allerdings 3 nicht benutzt werden. Der Wert der Integer-Variablen ist in der Reihenfolge High-/Lowbyte in den beiden auf den Namen folgenden Bytes abgelegt. Dies ist durch das Aussehen des FAC bedingt, in dessen Mantisse ebenfalls das höherwertige Mantissen-Byte vor dem niederwertigen steht.

Integerfelder unterscheiden sich nur in zwei Punkten in ihrem Aufbau von REAL-Feldern:

Zum einen natürlich im Variablennamen, in dessen beiden Zeichen die Bits 7 gesetzt sind.

Zum zweiten in der Länge der einzelnen Feldelemente, die beim Integerarray nur 2 statt 5 Byte beträgt.

5.5.3 String-Variable und -Felder

Für die Darstellung einer nicht dimensionierten String-Variablen werden wieder 7 Byte vereinnahmt. Wie bei den nicht dimensionierten Integer-Variablen ist der Grund dafür, daß eine einheitliche Variablenlänge eine schnellere Variablensuche erlaubt.

Die Reihenfolge der einzelnen Bytes einer String-Variablen ist:

1. Zeichen des Namens
2. Zeichen des Namens geodert mit $80

Länge des Strings
Lowbyte der String-Adresse im String-Bereich
Highbyte der Adresse

Stringarrays unterscheiden sich von den anderen Feldern durch die Länge der einzelnen Feldelemente, die hier gleich drei ist.

5.5.4 FN-Variable

FN-Variable, Variable, die durch einen BASIC-Befehl wie

```
DEF FN AB(X) = 2*X
```

angelegt werden, liegen etwas außerhalb des Rahmens der Variablentypen, die bisher betrachtet wurden. In solchen Variablen wird keine Zahl oder Zeichenkette festgehalten, sondern ein Ort im BASIC-Programmtext, die Stelle im Text nämlich, an der sich die FN-Definition, hier »2*X«, befindet. Dies ist auch der Grund, warum der DEF-Befehl nicht im Direktmodus funktioniert: ohne Programmtext keine FN-Variablen, die in den Text zeigen.

Der Aufbau einer FN-Variablen:

1. Zeichen des Variablennamens mit $80 geodert
2. Zeichen

Lowbyte der FN-Definition im Programmtext
Highbyte

Lowbyte der Adresse des Argumentes X. X kann eine REAL- oder eine INTEGER-Variable sein.
Highbyte

1. Zeichen der Argument-Variable als Zeiger für den Variablentyp des Argumentes. (INTEGER: Bit 7 gesetzt, REAL: Bit 7 = 0)

Wie immer ist die Gesamtlänge der Variablen 7 Byte. FN-Felder sind natürlich schon deshalb nicht möglich, weil eine Klammer wie bei der Dimensionierung sowieso immer als zum Argument der Funktion zugehörig angesehen würde.

5.5.5 Variable suchen und anlegen

5.5.5.1 Variable suchen

Um aus eigenen Maschinenprogrammen heraus Variable zu suchen, gibt es verschiedene Möglichkeiten. Die weitaus bequemste davon ist aber wohl die Benutzung derjenigen Routine, die auch der Interpreter für diesen Zweck benutzt:

FETVAR = $7978

Diese Routine übernimmt sowohl das Lesen und Auswerten des Variablennamens (Typerkennung, Erkennung dimensionierter Variablen), als auch den Transport der verschiedenen möglichen Variablenwerte in die entsprechenden Register in der Zeropage. Dazu ist es lediglich notwendig, daß der CHRGET-Programmzeiger auf dem ersten Zeichen des Variablennamens steht. Alle weiteren Zeichen holt sich FETVAR dann selbständig mit CHRGET.
Da FETVAR sowohl String-Variable als auch REAL- oder INTEGER-Variable holen kann, wobei diese auch noch dimensioniert sein können, benutzt FETVAR einige Flags in der Zeropage, die angeben, um welchen Variablentypus es sich gehandelt hat:

VALTYP = $0f Ist VALTYP gleich $ff, wurde eine String-Variable geholt, ist das Flag Null, war es eine numerische Variable.

INTFLG = $10 Im Fall, daß es eine numerische Variable war, unterscheidet dieses Flag zwischen INTEGER und REAL: Das Flag ist gleich $80 für INTEGER-, gleich Null für REAL-Variable.

Der Vollständigkeit halber sollen noch zwei weitere Flags erwähnt werden, die mittelbar mit FETVAR zu tun haben:

COUNT = $0d Zwischenspeicher für die Anzahl der Elemente eines Feldes, das von FETVAR gelesen wird. Stimmt diese Anzahl nicht mit der entsprechenden Anzahl im Feld-Header überein, gibt es einen Error »bad subscript«.

DIMFLG = $0e Flag für Aufruf von FETVAR durch den DIM-Befehl. Da FETVAR auch Variable und Felder anlegt, wird die Routine auch zur Dimensionierung von Feldern benutzt. Dazu wird einfach das höchste Feldelement bei DIMFLG ungleich Null angelegt.

Sehen wir uns jetzt einmal an, was FETVAR nach dem Aufruf zurückliefert:

In der Zeropage-Adresse

VARNAM = $47/ $48

befinden sich die beiden Zeichen des Variablennamens. In den beiden Zeichen sind, je nach Typ, die höchsten Bits gesetzt oder gelöscht. Ist der Variablenname nur ein Zeichen lang, wird als zweites Zeichen ein Nullbyte genommen.
Anhand von VALTYP und INTFLG können wir die verschiedenen Variablentypen unterscheiden.
In der Adresse

VARPNT = $49/ $4a

finden wir die Adresse der aktuellen Variablen in RAM-Bank 1 wieder. Diese Adresse weist hinter den Variablennamen im Variableneintrag, also weist sie auf das erste Zeichen des Variablenwertes beziehungsweise des String-Descriptors.
Im Fall, daß eine String-Variable geholt wurde, ist die Arbeit von FETVAR hiermit beendet.
Bei numerischen Variablen wird zusätzlich der Variablenwert nach FAC geholt. Integerwerte werden in das Fließkommaformat umgerechnet und normalisiert.
Interessant ist noch, daß FETVAR die verschiedenen Systemvariablen wie ST, DS$ usw. richtig interpretiert!

5.5.5.2 Variable anlegen/ Wertzuweisung an Variable

Auch den umgekehrten Fall, daß ein Wert in eine Variable gebracht werden soll, erledigen wir am bequemsten durch den Einsatz einer Interpreterroutine:

GETVAR = $7aaf

GETVAR ist eine Unterroutine von FETVAR, die in allen Punkten mit FETVAR bis auf das Holen der Variablenwerte übereinstimmt. GETVAR ist also mit der Bereitstellung des Zeigers VARPNT auf den Variablenwert beendet.

Dies ist notwendig, wenn einer Variablen ein Wert zugewiesen werden soll, denn durch FETVAR würden ja die Register, die den neuen Variablenwert enthalten, überschrieben. An dieser Stelle ein Wort dazu, wie Variable eigentlich angelegt werden. Anlegen von Variablen meint: Wie gelangt eigentlich der Variablen-Descriptor mit dem Variablennamen in die Bank 1?

Zunächst einmal: Es gibt keine eigene Routine zum Anlegen von Variablen. Kann FETVAR oder GETVAR eine Variable im Variablenraum nicht finden, so wird diese neu angelegt, das heißt ein entsprechender Eintrag im Variablenfeld wird erzeugt. Beim Anlegen einer nicht dimensionierten Variablen werden dazu die Felder um 7 Byte nach oben verschoben. Neu angelegte Variable haben immer den Startwert Null.

Bleibt noch das Problem, wie man einen Wert beispielsweise von FAC mit möglichst wenig Aufwand in eine Variable praktiziert. Ob eine solche Variable schon vorhanden war oder nicht, braucht uns nicht zu kümmern, da GETVAR die Erzeugung der neuen Variablen automatisch vornimmt.

Dazu muß noch ein Hilfszeiger erwähnt werden, der für den Transport des Variablenwertes in das Variablenfeld benötigt wird:

LSTPNT = $4b/ $4c

Dieser Zeiger muß vor dem Werttransport durch die weiter unten angeführten Routinen mit dem Wert von VARPNT gefüllt werden, da diese Routinen den Zeiger benötigen. Normalerweise kann man dies am besten gleich nach dem Aufruf von GETVAR vornehmen, da Akku und Y-Register beim rts von GETVAR den Wert von VARPNT enthalten:

```
sta lstpnt
sty lstpnt+1
```

Die einzelnen Zuweisungsroutinen:

LETINT = $53e5	FAC wird gerundet, nach INTEGER gewandelt und in der durch LSTPNT bezeichneten Variablen abgelegt.
LETREAL = $53fa	FAC wird gerundet und in einer REAL-Variablen abgelegt.
LETSTR = $5405	String-Zuweisung. Hier sind verschiedene Systemvariablen zu berücksichtigen, beispielsweise ti$. Diese möchte ich an dieser Stelle einmal vernachlässigen. Es bleiben dann noch zwei Fälle zu unterscheiden:

Fall 1: Der String-Variablen wird der String aus einer anderen Variablen zugewiesen:

```
a$ = b$
```

Der Zeiger auf den zuzuweisenden String-Descriptor muß in der Zeropage-Adresse VARPNT übergeben werden.

Fall 2: Der String-Descriptor befindet sich im temporären Stringstack:

```
a$ = "string"
```

In diesem Fall gilt der letzte Eintrag in den Stringstack als Referenz. LASTPT = $19/ $1a weist auf diesen Eintrag. Durch die Zuweisung wird gleichzeitig der Eintrag aus dem Stringstack entfernt.

5.6 Interpreterroutinen zur Auswertung beliebiger Ausdrücke

Wir haben zwar jetzt eine ganze Menge einzelner Routinen kennengelernt, eine ganz wichtige Sache fehlt aber noch: Was macht der Interpreter, um einen Ausdruck wie in

```
print (a+2)*7↑(1-sin(x))
```

zu berechnen? In einem solchen Ausdruck müssen Variable geholt, müssen Konstante in Fließkommazahlen verwandelt werden und nicht zuletzt muß auch die Ausführung der einzelnen Operationen in der richtigen Reihenfolge stattfinden, damit die Klammern berücksichtigt werden.

Keine Angst, das brauchen wir glücklicherweise nicht selbst zu übernehmen. Im Interpreter existiert eine Routine, die ganz allein die Auswertung völlig beliebiger Ausdrücke übernimmt:

FRMEVL = $af96

Als einzigen Eingangsparameter benötigt diese Routine den CHRGET-Programmzeiger TXTPTR. Dieser muß auf dem ersten Zeichen des auszuwertenden Ausdrucks stehen. Im Beispiel also auf der geöffneten Klammer.
Als Ergebnis nach Aufruf von FRMEVL erhält man zunächst das schon bekannte Flag VALTYP. Dieses Flag zeigt an, ob der Ausdruck ein String war oder ob es sich um einen numerischen Ausdruck gehandelt hat.
War es ein numerischer Ausdruck, befindet sich der ermittelte Wert in FAC. Da alle Rechnungen des Interpreters im Fließkommaformat erfolgen, ist dieses Ergebnis ebenfalls immer eine Fließkommazahl.
Wurde ein String-Ausdruck geholt, ist der Einsatz einer zweiten Routine notwendig, um sich die Parameter des Strings zu verschaffen:

STRPAR = $877e

Beide Routinen, FRMEVL und STRPAR, sind auch zusammengefaßt in

FRESTR = $877b FRMEVL String

Darin ist gleichzeitig eine Fehlerabfrage enthalten, die ein »type mismatch« ausgibt, wenn das FRMEVL-Ergebnis nicht vom Typ String ist. Die Adresse des Strings erhält man in

INDEX = $24/ $25

zurück, die Länge des Strings im Akkumulator.

5.7 Weitere Routinen des Interpreters

In diesem Kapitel sollen einige Routinen des Interpreters aufgeführt werden, deren Entsprechungen sich schon beim C64 großer Beliebtheit erfreut haben.
Ganz im Gegensatz zum Betriebssystem, das ja geradezu dazu geschrieben wurde, damit seine Routinen von anderen Programmen benutzt werden, sind die meisten Routinen des Interpreters nicht oder nur schwer für eigene Zwecke nutzbar. BASIC ist eben ein Anwenderprogramm, dessen Routinen im allgemeinen BASIC-spezifisch sind.

Den Routinen der Fließkomma-Arithmetik wurde schon ein eigenes Kapitel gewidmet. Die übrigen hier aufgeführten Routinen dienen den unterschiedlichsten Aufgaben.

FRMEVL = $af96

Auswertung eines beliebigen Ausdrucks
IN: TXTPTR auf erstem Textzeichen des Ausdrucks
FRMEVL wurde schon vorgestellt. Die Routine arbeitet beliebig aufgebaute Ausdrucke ab. Die einzelnen Textzeichen des Ausdrucks werden mit CHRGET geholt. Nach Verlassen der Routine steht der CHRGET-Programmzeiger auf dem ersten Zeichen hinter dem Ausdruck. Der Werttyp des geholten Ausdrucks wird in VALTYP = $0f angezeigt.
OUT: VALTYP
FAC bei numerischen Ausdrucken; bei Strings wird STRPAR aufgerufen, um die Parameter des Ergebnis-Strings zu holen.

FRMNUM = $77d7

Holen eines numerischen Ausdrucks
IN: wie FRMEVL
FRMNUM holt mit Hilfe der Routine FRMEVL einen numerischen Ausdruck nach FAC. Dazu wird der Werttyp VALTYP nach Ablauf von FRMEVL geprüft. War der geholte Ausdruck ein String, wird die Fehlermeldung »type mismatch« ausgegeben.
OUT: FAC

STRPAR = $877e

IN: Von FRMEVL geholter String-Ausdruck
STRPAR holt die Parameter eines von FRMEVL geholten String-Ausdrucks. Die Adresse des Ergebnis-Strings erscheint in der Zeropage, Adresse $24/ $25 sowie im X- und Y-Register. Die Länge des Strings wird im Akku übergeben.
OUT: A = Länge des Strings
X = Lowbyte der Adresse
Y = Highbyte der Adresse

FRESTR = $877b

FRMEVL mit Prüfung auf String und Holen der String-Parameter
FRESTR entspricht der Routine FRMNUM, nur, daß FRESTR speziell für das Holen von Strings zuständig ist. Stimmt der VALTYP nicht mit dem erwarteten String-Typ überein, gibt es wieder ein »type mismatch«. Die String-Parameter werden wie bei STRPAR übergeben.
OUT: siehe STRPAR

CHKSTR = $77dd

Prüfung VALTYP auf String

CHKNUM = $77da

Prüfung VALTYP auf numerisch
Beide Routinen verzweigen bei Nichteinhaltung des richtigen Werttyps zur Ausgabe von »type mismatch«. Ansonsten kehren sie einfach zum aufrufenden Programm zurück.

GETBYT = $87f1

Holen eines Wertes zwischen 0 und 255
GETBYT holt mit Hilfe der FRMNUM-Routine einen Byte-Wert in das X-Register. Ergibt sich ein Wert größer als 255, wird der Fehler »illegal quantity« ausgegeben. Bei Aufruf dieser Routine muß der Programmzeiger vor dem ersten Zeichen des Ausdrucks stehen.
OUT: X = Bytewert
A = letztes gelesenes Zeichen

GETADR = $8812

Holt eine positive Adresse
IN: TXTPTR auf erstem Zeichen des Ausdrucks
Die Routine holt mit FRMNUM einen numerischen Ausdruck nach FAC. Dieser darf nicht negativ oder größer als $ffff sein, sonst gibt es eine Fehlermeldung »illegal quantity«. Die Fließkommazahl in FAC wird nach INTEGER gewandelt und in die Adresse

LINNUM = $16/ $17

gebracht. Sie erscheint auch in den Prozessorregistern.
OUT: A = Highbyte des Integerwertes
Y = Lowbyte
dasselbe auch in FAC und LINNUM
COM: Die Adresse GETADR-3 schließt eine Prüfung auf ein führendes Komma mit ein. Damit können dann Ausdrücke wie zum Beispiel
, 2048
geholt werden.

POKADR = $8803

Holt eine Adresse und einen durch Komma getrennten Byte-Wert wie beim POKE-Befehl. Die Adresse erscheint in LINNUM, der Byte-Wert im X-Register.

CHKKOM = $795c

Prüfung auf Komma im Programmtext.
IN: TXTPTR
Das Zeichen, auf das TXTPTR aktuell weist, wird mit dem Komma verglichen. Bei Nicht-

übereinstimmung wird ein »syntax error« ausgegeben, ansonsten wird das nächste Zeichen hinter dem Komma mit CHRGET gelesen.
OUT: A = Zeichen hinter dem Komma

CHKSGN = $795e

Prüfung auf beliebiges Zeichen im Text
IN: TXTPTR
A = Vergleichszeichen
Es wird das Zeichen, auf das der TXTPTR weist, mit einem im Akku befindlichen Zeichen verglichen. Ist die Prüfung positiv verlaufen, wird das nächste Textzeichen mit CHRGET geholt und zum aufrufenden Programm zurückgekehrt. Ansonsten gibt es wieder einen »syntax error«.
OUT: A = nächstes Zeichen

DFTBYT = $9e1c

Byte-Wert holen, wenn noch ein Ausdruck folgt, sonst Default-Wert = 0
IN: TXTPTR auf erstem Zeichen des erwarteten Ausdrucks
DFTBYT holt Parameter aus einer Parameterliste, die etwa so aussehen kann:
123,,2,,,2
Solche Parameterlisten werden im Interpreter häufiger verwendet, wobei zwei Kommas hintereinander bedeuten sollen, daß der entsprechende Parameter, der an sich zwischen Kommas zu stehen kommen würde, auf einen Default-Wert gesetzt werden soll, der im allgemeinen Null ist. DFTBYT prüft zunächst, ob schon das Ende der Parameterkette erreicht ist. Ist dies der Fall, wird das Carry als Flag für den Default-Wert gelöscht und zurückgesprungen. Ist noch kein Nullbyte oder Doppelpunkt erreicht, wird geprüft, ob ein Komma folgt. Ist dies nicht der Fall, ist die Parameterliste nicht korrekt weitergeführt und es gibt einen »syntax error«. Ist die Liste korrekt mit einem Komma fortgesetzt, wird getestet, ob noch ein Ausdruck oder ob direkt ein zweites Komma folgt. Wird das Komma gefunden, wird die Routine wieder mit gelöschtem Carry als Flag für den Default-Wert verlassen.
OUT: X= Byte-Wert
clc = Flag für Default-Wert 0 in X
sec = Flag für kein Default
COM: DFTBYT+2 ist dieselbe Routine, wobei man im X-Register einen eigenen Default-Wert übergeben kann.

DFTADR = $9e06

Default-Wert oder Adresse holen
IN: TXTPTR wie bei DFTBYT
DFTADR leistet dasselbe wie DFTBYT für das Holen von Adressen. Das Carry dient wieder als Flag dafür, ob ein Default-Wert oder ein gelesener Wert übergeben wird.

OUT: A = Highbyte einer Adresse
Y = Lowbyte
clc = Default-Wert 0 in A und Y
sec = gelesener Wert in A und Y

STROUT = $55e2

Ausgabe eines Strings
IN: A = Lowbyte der Startadresse des Strings
Y = Highbyte
Der auszugebende String muß mit einem Nullbyte abgeschlossen sein, da STROUT die String-Länge selbst ermittelt und den String in den String-Bereich der Bank 1 kopiert. Die Routine enthält leider einen winzigen Fehler, der in ähnlicher Form schon beim 64er festzustellen war. Bei logischen File-Nummern größer 128 wird leider kein Line Feed (ASCII 10) zusätzlich zum Carriage Return ausgegeben.
OUT: -

CROUT = $5598

Ausgabe von Carriage Return und gegebenenfalls Line Feed
IN: -
Den obigen Fehler kann man eventuell ausbügeln, indem das Return mit CROUT ausgegeben wird. In CROUT wird in Abhängigkeit von der logischen File-Nummer auch das Line Feed erzeugt.
OUT: -

DECOUT = $8e32

Ausgabe einer Dezimalzahl von 0 bis 65535
IN: A = Highbyte der Zahl
X = Lowbyte
Die übergebene Integerzahl wird in das Fließkommaformat gewandelt und dann in einen String in FBUFFR ($100) übersetzt. Dieser String wird ausgegeben. BASIC nutzt diese Routine beispielsweise bei der Ausgabe von Zeilennummern.

GETLIN = $4f93

Holen einer Eingabezeile in den Eingabepuffer ab $200
IN: -
Mit GETLIN holt der Interpreter Eingabezeilen im Direktmodus oder beim INPUT-Befehl. Der Cursor erscheint auf dem Bildschirm, wenn der Bildschirm gerade das aktuelle Eingabegerät ist. Die Eingabe ist dann beendet, wenn die Return-Taste (oder Control + m) gedrückt wird. Die gelesene Eingabezeile wird im Puffer

BUF = $200

abgelegt und mit einem Nullbyte als Endekennzeichen abgeschlossen. Die Startadresse des Puffers wird in X- und Y-Register übergeben, so daß der CHRGET-Programmzeiger TXTPTR sofort auf den Puffer gesetzt werden kann.

OUT: BUF = Eingabezeile
X = Lowbyte der Startadresse des Puffers
Y = Highbyte Start des Puffers

OFFSET = $52a7 (OFFDOP = $52a2; OFFNIL = $52a5)

Ermitteln des Offsets zu einem Suchzeichen (Doppelpunkt, Nullbyte)

IN: X = Suchzeichen bei OFFSET (andere: -)

Ausgehend von der Position des TXTPTR wird in einer Programmzeile nach einem Suchzeichen (Trennzeichen) gesucht. Die Suche wird bei Erreichen des Zeilenendes abgebrochen.

OUT: Y = Offset von TXTPTR

OFFADD = $5292

Addition des Y-Registers zum TXTPTR

NXTSTA = $528f

Programmzeiger auf nächstes BASIC-Statement richten, also hinter den nächsten Doppelpunkt oder hinter das Zeilenende.

DOSPAR = $a3c3

Parameter für DLOAD, DSAVE,... holen

IN: A = Syntaxcheckbyte 1
X = Syntaxcheckbyte 2

Diese Routine ist sehr allgemein gehalten und eignet sich deshalb auch gut für eigene Anwendungen. Ihre Aufgabe besteht in der Auswertung einer allgemeinen Parameterliste, wie sie bei DLOAD, DSAVE, DOPEN oder auch SCRATCH und DIRECTORY vorkommen kann. Um die vielen verschiedenen erlaubten Parameterlisten auf ihre Gültigkeit überprüfen zu können, werden zwei Syntaxcheckbyte benutzt:

SYNTAX1 = $80

und

SYNTAX2 = $81

Einzelne Bits dieser beiden Flags haben die folgende Bedeutung:

SYNTAX1:

Bit	Bedeutung
0	1. File-Name gelesen
1	2. File-Name gelesen
2	logische File-Nummer gelesen
3	Unit (Gerätenummer) gelesen
4	1. Laufwerk gelesen
5	2. Laufwerk gelesen
6	Recordlänge gelesen
7	Klammeraffe gelesen (Überschreiben von Files)

SYNTAX2:

Bit	Bedeutung
0	Segmentnummer gelesen
1	Startadresse gelesen
2	Endadresse gelesen

Nach Aufruf von DOSPAR sind einzelne Bits der beiden Syntax-Flags gesetzt. Damit wird angezeigt, daß das Ereignis, das dem Bit zugeordnet ist, eingetreten ist. Beispiel:
Die beiden Bits sollen nach Aufruf von DOSPAR die Werte

SYNTAX1 = %1000 0001

und

SYNTAX2 = %0000 0110

haben. Daran läßt sich ablesen, daß eine Parameterliste wie

```
"@filename",p2000 to 4000
```

ausgewertet wurde.

Bei Aufruf von DOSPAR werden im Akkumulator und im X-Register zwei den Syntax-Flags entsprechende Byte übergeben, durch die ein Syntax-Check schon innerhalb der DOSPAR-Routine erfolgt. Ein gesetztes Bit in diesen beiden Byte verbietet das entsprechende Ereignis. Übergebe ich im Akku beispielsweise den Wert

A = %0000 1000

so verbiete ich damit das Holen eines Parameters der Form »u9«, mit dem man die Gerätenummer eingibt. Taucht ein Parameter dieser Form in der Parameterliste doch auf, gibt es einen »syntax error«.

Von Interesse ist nun noch, wohin die einzelnen gelesenen Parameter geschrieben werden: dazu dienen die ersten 18 Byte des Stack-Bereiches hinter dem String-Puffer der Arithmetik:

DOSBUFFR = $111 bis $122

Die Zuordnung der einzelnen Bytes, in Klammern jeweils eine Angabe, wie der dem Byte entsprechende Parameter in der Parameterliste aussieht:

DOSF1L	**= $111**	Länge des 1. Namens, der Name selbst ist nach SAVRAM = $12b7 kopiert worden
DOSF2L	**= $113**	Länge des 2. Dateinamens
DOSF2A	**= $115/ $116**	Adresse des 2. Namens in Bank 1
DOSDS1	**= $112**	Laufwerk 1 (0 oder 1), d 1
DOSDS2	**= $114**	Laufwerk 2 (0 oder 1)
DOSLA	**= $11b**	logische Adresse (# 1)
DOSFA	**= $11c**	Gerätenummer (u 8)
DOSBNK	**= $11f**	Konfigurationsindex (on b1)
DOSOFL	**= $117/ $118**	Startadresse (p 2000)
DOSOFH	**= $119/ $11a**	Endadresse (to 4000)
DOSDID	**= $120/ $121**	2 Zeichen ID (i 2a)
DIDCHK	**= $122**	Flag für ID eingegeben ($ff)
DODRCL	**= $11e**	Recordlänge (l 128)

Bit 6 des Flags SYNTAX1 wird gesetzt, wenn eine relative Datei (l) oder eine Schreibdatei (w) gewählt wurde.

Das Byte

DOSSA	**= $11d**	dient als Speicher für eine verwendete Sekundäradresse

Gerätenummer, logische File-Nummer, Sekundäradresse sowie Start- und Endadresse erhalten Default-Werte, wenn diese nicht ausdrücklich angegeben werden:

Gerät = 8
logische File-Nummer = 0
Sekundäradresse = 15 + $60
Start- und Endadresse = $ffff

COMCPL = $a667
Erzeugung eines Kommando-Strings aus den mit DOSPAR geholten Parametern
IN: A = Term-Zähler
Y = Kommandoschlüssel

Offensichtlich reicht die Arbeit von DOSPAR noch nicht aus, um mit einer Floppy tatsächlich arbeiten zu können. Es fehlt noch das Zusammensetzen der einzelnen Parameter zu einem Ausgabe-String, den die Floppy verdauen kann. Beispielsweise muß eine Laufwerksnummer an den Anfang des Dateinamens gebracht werden, wie wir es von DLOAD gewohnt sind:

```
"0:name"
```

Oder es muß bei SCRATCH die Übersetzung erfolgen:

```
"s0: name"   etc.
```

Dieses Zusammenfügen der einzelnen Parameter übernimmt COMCPL. Um die gestellte Aufgabe erfüllen zu können, erhält COMCPL beim Aufruf zwei Parameter, die in eine ROM-Tabelle mit den möglichen Befehlen weisen. Je nach diesen Tabellenzeigern wird der Kommando-String zusammengefügt.
Da COMCPL sowieso nur in der Lage ist, die in BASIC implementierten Kommandos zu erzeugen, möchte ich mich hier darauf beschränken, empirisch aufzuzählen, zu welchen Kommandos welche Term-Zähler und Kommandoschlüssel gehören:

Befehl	Term-Zähler	Schlüssel
dclear	2	255
directory	1 ohne Laufw. und Name	1
	2 ohne Name	
	4 mit Name und Laufw.	
dsave etc	4	5
append	5	22
header	6 mit, 4 ohne ID	27
collect	1 ohne Laufw., mit :2	33
backup	4	35
copy	8	39
rename	8	47
scratch	4	55
record	4	59

Der von COMCPL erzeugte Kommando-String erscheint in

DOSSTR = $1100

OUT: DOSSTR = Kommando-String

MISTAKE = $4d76
Ausgabe der BASIC-Fehlermeldungen
IN: X = Nummer des Fehlers
Arbeiten eigene Maschinenprogramme mit dem BASIC-Interpreter zusammen, möchte man natürlich auch entsprechende Fehlermeldungen wie der Interpreter ausgeben können. Dies ist mit der MISTAKE-Routine möglich. Gleichzeitig wird der RUN-Modus durch MISTAKE ausgeschaltet und nach Ausgabe der Fehlermeldung in die Warteschleife für den Direktmodus übergegangen (READY). MISTAKE wird immer mit einem jmp angesprungen. Die Zuordnung der Fehlernummern zu den Meldungen:

Nummer	Meldung
1	too many files
2	file open
3	file not open
4	file not found
5	device not present
6	not input file
7	not output file
8	missing file name
9	illegal device number
10	next without for
11	syntax
12	return without gosub
13	out of data
14	illegal quantity
15	overflow
16	out of memory
17	undef'd statement
18	bad subscript
19	redim'd array
20	division by zero
21	illegal direct
22	type mismatch
23	string too long
24	file data
25	formula too complex
26	can't continue
27	undef'd function
28	verify
29	load
30	break

31	can't resume
32	loop not found
33	loop without do
34	direct mode only
35	no graphics area
36	bad disk
37	bend not found
38	line number too large
39	unresolved reference
40	unimplemented command
41	file read

SETVEC = $4251

BASIC-Vektoren aus dem ROM in die erweiterte Zeropage kopieren
IN: –
Von SETVEC werden alle BASIC-Vektoren aus dem ROM ins RAM kopiert. Diese Routine ist vor allem dann von Nutzen, wenn BASIC-Erweiterungen, die durch Änderung der BASIC-Vektoren in den Interpreter eingeschleift sind, ausgeschaltet werden sollen.
OUT: –

SEARCH = $43e2

Suchen eines Befehlswortes in einer Tabelle
IN: A = Highbyte der Tabellenstartadresse
Y = Lowbyte der Adresse
Der Zeropage-Pointer INDEX = $24/ $25 wird als Hilfszeiger in eine Tabelle von Befehlsworten benutzt. Der Programmzeiger TXTPTR = $3d muß auf dem ersten Zeichen des zu identifizierenden Befehlswortes stehen. Die Tabelle muß einen bestimmten Aufbau haben, damit SEARCH funktionieren kann: Die einzelnen Worte der Tabelle werden hintereinander geschrieben, wobei jeweils das letzte Zeichen des Befehlswortes zum Großbuchstaben gemacht wird. Auf diese Weise wird das Ende eines einzelnen Befehlswortes erkannt. Das Ende der Tabelle wird mit einem Nullbyte markiert.
OUT: Minus-Flag gesetzt, wenn das Wort in der Tabelle gefunden wurde
A = Nummer des Wortes in der Tabelle mit $80 geodert
COM: Der Akkumulator findet sich auch in COUNT = $0d wieder
Das X-Register bleibt unverändert

USTLST = $516a

Listen eines durch Usertoken eingeführten Befehlswortes
IN: A = Highbyte der Tabelle der neuen Befehlsworte
Y = Lowbyte der Tabelle
X = Nummer des Befehlswortes in der Tabelle plus $80

Genau wie SEARCH aus einer speziell aufgebauten Tabelle die Nummer eines Wortes in der Tabelle ermitteln kann, kann USTLST das Wort über BSOUT ausgeben, wenn die Nummer des Tabelleneintrags und die Adresse der Tabelle übergeben wird. Beide Routinen werden benötigt, wenn eigene Befehle über Usertoken in den Interpreter eingeschleift werden.
OUT: –

LINNUM = $50a0

Holt Zeilennummer nach $16/ $17 in der Zeropage
IN: A = erste Ziffer
Die Routine LINNUM, nicht zu verwechseln mit der Zeropage-Adresse LINNUM = $16/ $17, holt eine Integeradresse in die Zeropage. Dabei arbeitet LINNUM ohne Unterstützung der Routine FRMEVL! BASIC benutzt die LINNUM-Routine, um Zeilennummern bei der Eingabe von Programmzeilen zu lesen, da LINNUM ausschließlich Ziffernfolgen und keine arithmetischen Ausdrücke verarbeitet.
OUT: $16/ $17 = Zeilennummer Low-/ Highbyte

LINSEA = $5064

Suchen einer Programmzeile
IN: $16/ $17 = Zeilennummer
Zu einer gegebenen Zeilennummer sucht LINSEA die Adresse der Zeile, falls eine Zeile mit der Zeilennummer vorhanden ist. LINSEA wird zum Beispiel beim GOTO oder GOSUB eingesetzt.
OUT: clc = Zeile mit der Nummer gibt's nicht
sec = Zeile gefunden
LOWTR = $61/ $62: steht auf Lowbyte des Linkers der gefundenen Zeile

5.8 Beispiele

Nach den etwas trockenen Exkursionen durch den Interpreter sollen jetzt wieder einige mehr oder weniger praktische Beispiele folgen, an denen man den Umgang mit den Interpreterroutinen kennenlernen kann.

Beispiel 1:

Als erstes und gleichzeitig einfachstes Beispiel soll eine Eingabezeile vom Bildschirm geholt werden, die Eingabe wollen wir dann auswerten und als Kontrolle, ob alles korrekt abläuft, gleich wieder ausgeben:

```
                .base $1300         ; Startadresse

  .define txtptr   = $3d            ; CHRGET-Programmzeiger
  .define chrget   = $380           ; CHRGET-Routine
  .define getlin   = $4f93          ; Eingabezeile nach BUF=$200 holen
  .define crout    = $5598          ; Carriage Return ausgeben
  .define strout   = $55e2          ; Stringausgabe
  .define frmnum   = $77d7          ; numerischen Ausdruck holen
  .define factostr = $af06          ; FAC in ASCII-String wandeln
  start      jsr crout              ; Zeilenvorschub
             jsr getlin             ; Eingabezeile holen: Cursor blinkt
             stx txtptr             ; Programmzeiger auf Start des
             sty txtptr+1           ; Eingabepuffers richten
             jsr chrget             ; TXTPTR auf 1. Zeichen des Ausdr.
             jsr frmnum             ; numerischen Ausdr. holen
             jsr factostr           ; FAC nach String wandeln
             jsr strout             ; Zahlen-String ausgeben
             jmp start              ; Schleife
```

Ich bin fast sicher, Sie hätten nicht erwartet, daß die Sache so einfach aussieht, wenn sie einmal niedergeschrieben ist. Die einzelnen Routinenaufrufe stehen ja praktisch so hintereinander wie man spricht. Dies ist natürlich kein Zufall, denn welchen Grund sollten die Autoren des BASIC 7.0 gehabt haben, sich das Leben selbst schwer zu machen. Die Routinen sind deswegen weitgehend aufeinander abgestimmt. So übergibt die Routine FACTOSTR gerade in Akku und Y-Register die Zeiger auf den String, die STROUT benötigt. Auch bei vielen anderen Routinen ist das in ähnlicher Weise der Fall, wie wir noch sehen werden.

Das obige Beispielprogramm kann mit

```
bank 15 : sys dec("1300")
```

oder vom Monitor aus mit

```
j f1300
```

gestartet werden. Blinkt der Cursor, kann man einen beliebigen Ausdruck eingeben, der nach Abschluß der Eingabe durch die Return-Taste von FRMEVL (FRMNUM) berechnet wird. Das Ergebnis der Rechnung in FAC wird in einen Zahlen-String gewandelt und auf dem Bildschirm ausgegeben. Das Ganze arbeitet also ähnlich dem PRINT-Befehl.

Beispiel 2:
Wir wollen das Ergebnis der Berechnung einmal nicht in der gewohnten dezimalen Form ausgeben, sondern den FAC byteweise wie bei einem Hexdump ausgeben. Auf diese Weise können wir auch ermitteln, wie Fließkomma-Konstante aussehen müssen, die wir in eigene Programme einbauen wollen. Dazu ist nur zu beachten, daß sich das Zeropage- und das Speicherformat einer Fließkommazahl in der Darstellung des Mantissenvorzeichens unterscheiden.

```
          .base $1300

.define txtptr   = $3d    ; CHRGET-Programmzeiger
.define fac1     = $63    ; FAC
.define chrget   = $380   ; CHRGET-Routine
.define getlin   = $4f93  ; Eingabezeile holen
.define crout    = $5598  ; Zeilenvorschub
.define frmnum   = $77d7  ; numerischen Ausdr. holen
.define cr       = $ff00  ; Konfigurationsregister
.define bsout    = $ffd2  ; Kernal-BSOUT

start     jsr crout       ; neue Zeile
          jsr getlin      ; Eingabezeile holen
          stx txtptr      ; TXTPTR auf Pufferstart
          sty txtptr+1    ; richten
          jsr chrget      ; TXTPTR auf 1. Zeichen
          jsr frmnum      ; Ausdruck holen
          jsr facout      ; byteweise ausgeben
          jmp start       ; Schleife
facout    jsr crout       ; neue Zeile
          ldx #0          ; Zähler
          stx cr          ; I/O einschalten !!!
m1        lda fac1,x      ; Byte aus FAC
          jsr hexout      ; sedezimal ausgeben
          lda #32         ; Leerzeichen
          jsr bsout       ; dazwischen
          inx             ; Zähler: +1
          cpx #5          ; schon 5 Byte?
          bcc m1          ; nein: Schleife
          rts             ; ja: fertig
hexout    pha             ; Byte auf Stack retten
          lsr             ; /2
          lsr             ; /4
```

```
         lsr              ; /8
         lsr              ; /16 = oberes Halb-Byte
         jsr nibout       ; ausgeben
         pla              ; Byte zurückholen
         and #15          ; unteres Halb-Byte isolieren
nibout   cmp #10          ; noch als Dez.-Ziffer darzust.?
         bcc nb           ; ja: Sprung
         adc #6           ; sonst: 7 addieren (6 + Carry)
nb       adc #$30         ; nach ASCII
         jmp bsout        ; und ausgeben
```

Starten wir diese Routine einmal wie die vorhergegangene, so erhalten wir als Ausgabe fünf Byte nebeneinander so dargestellt, wie wir es von einer Hexdumpzeile des Monitors gewohnt sind. Das erste Byte ist der Exponent von FAC, dahinter folgen die 4 Mantissen-Byte, wobei man sich hinter dem Exponenten-Byte den Dezimalpunkt oder Binärpunkt vorzustellen hat.
Geben wir als Ausdruck einmal einfach die Zahl 1 ein, erhalten wir folgende Ausgabe:

81 80 00 00 00

Dies entspricht

% 0.1000... * 2 (129-128) = 1/2 * 2 = 1

da ja immer 128 zum Exponenten hinzuaddiert ist. Vielleicht probieren Sie einmal die weiteren Zahlen aus?

Das vorliegende kleine Beispiel gibt auch noch einmal Anlaß, einen sehr wichtigen Punkt erneut anzusprechen: die I/O in Zusammenhang mit der Verwendung von Interpreterroutinen. Besonders diejenigen, die schon auf dem C64 gearbeitet haben, werden sich etwas umstellen müssen: Eine einfache Zeichenausgabe, gleichgültig was man gerade unternimmt, ist nicht mehr ohne weiteres möglich.

Sehen Sie sich zur Erläuterung dieser These einmal die CHRGET-Routine noch einmal genau an, und zwar im Hinblick darauf, was mit dem Konfigurationsregister geschieht, speziell, wie der I/O-Bereich geschaltet wird: Man stellt fest, daß das Beschreiben des LCRC am Ende der CHRGET-Routine den I/O-Bereich abschaltet! Dies heißt nichts anderes, als daß jedesmal, wenn wir eine Interpreterroutine aufrufen, mit fast an Sicherheit grenzender Wahrscheinlichkeit der I/O-Bereich abgeschaltet ist. Wir müssen also immer dafür sorgen, daß die I/O gesondert von uns eingeschaltet wird, bevor wir Ausgaben vornehmen! Dasselbe gilt natürlich auch für Eingaben, die wir erwarten. Die Konsequenz, wenn dies nicht beachtet wird, ist, daß der unter der I/O liegende Programmtext zerstört wird und außerdem unsere Ein- oder Ausgaben nicht funktionieren. Sie können

dies auch einmal ausprobieren, indem Sie den Befehl »stx cr« im obigen Beispiel weglassen. Wenn Sie einen 80-Zeichen-Monitor haben, werden Sie sich wundern, wo die Ausgaben bleiben!

Beispiel 3:
Auch das Rechnen mit der Fließkomma-Arithmetik ist ähnlich einfach wie im letzten Beispiel. Die entsprechenden Routinen sind so aufeinander zugeschnitten, daß sie einfach hintereinander aufgerufen werden können, wenigstens in den meisten Fällen. Man sollte sich schon zuerst davon überzeugen, daß dies tatsächlich möglich ist.
Als einfaches Beispiel möchte ich das Quadrat einer eingegebenen Zahl berechnen und ausgeben lassen:

```
          .base $1300

.define txtptr     = $3d       ; CHRGET-Zeiger
.define chrget     = $380      ; CHRGET-Routine
.define getlin     = $4f93     ; Eingabezeile holen
.define crout      = $5598     ; Zeilenvorschub
.define strout     = $55e2     ; String ausgeben
.define frmnum     = $77d7     ; numerischen Ausdr. holen
.define mularg     = $8a2c     ; FAC := FAC * ARG
.define roundf     = $af4b     ; FAC runden
.define factoa     = $af6c     ; ARG := FAC
.define factostr   = $af06     ; FAC in ASCII-String wandeln

start     jsr crout            ; neue Zeile
          jsr getlin           ; Eingabezeile holen
          stx txtptr           ; TXTPTR auf Start
          sty txtptr+1         ; der Eingabezeile
          jsr chrget           ; 1. Zeichen des Ausdrucks
          jsr frmnum           ; Ausdruck holen
          jsr factoa           ; nach ARG transportieren
          jsr mularg           ; FAC mit ARG multiplizieren
          jsr roundf           ; FAC runden
          jsr factostr         ; in String wandeln
          jsr strout           ; ausgeben
          jmp start            ; Schleife
```

Hier ist zu beachten, daß FAC vor der Umwandlung in einen ASCII-String gerundet werden muß, da bei Rechnungen im allgemeinen ein Overflow in Form des Rundungs-Bytes FACOV auftritt. Dieselbe Rundung ist notwendig, wenn der FAC nach Rechnungen in den Speicher oder in andere Register transportiert wird.

Beispiel 4:
In diesem Beispiel wollen wir ein wenig mit Variablen spielen. Es soll ein String eingegeben werden, der von uns einer String-Variablen zugewiesen wird. Nach Ablauf der Routine kann diese String-Variable dann mit

```
print a$
```

abgefragt werden, um zu prüfen, ob die Zuweisung geklappt hat.

```
        .base $1300

 .define txtptr    = $3d      ; CHRGET-Zeiger
 .define lstpnt    = $4b      ; Transportzeiger
 .define chrget    = $380     ; CHRGET-Routine
 .define getlin    = $4f93    ; Eingabezeile holen
 .define letstr    = $5405    ; String-Zuweisung
 .define crout     = $5598    ; Return ausgeben
 .define getvar    = $7aaf    ; Variable suchen/ anlegen
 .define frestr    = $877b    ; FRMEVL für Strings

 start     jsr crout          ; neue Zeile
           jsr getlin
           stx txtptr
           sty txtptr+1
           jsr chrget         ; 1. Zeichen des Strings
           jsr frestr         ; String holen
           lda txtptr+1       ; TXTPTR retten
           pha
           lda txtptr
           pha
           lda #< (vnam)      ; TXTPTR auf Variablennamen
           sta txtptr         ; richten
           lda #> (vnam)
           sta txtptr+1
           jsr getvar         ; Variable suchen/ anlegen
           sta lstpnt         ; Zeiger auf Variable in den
           sty lstpnt+1       ; Transportzeiger
           jsr letstr         ; String an V. zuweisen
           pla                ; TXTPTR zurückholen
           sta txtptr
           pla
```

```
            sta txtptr+1
            rts                 ; Rückkehr

vnam        .byte "a$"          ; Variablenname
```

Starten Sie das Programm wieder mit einem SYS oder mit dem J-Befehl des Monitors und geben Sie dann einen String ein. Sie sehen, die Zuweisung klappt problemlos.

Zu erläutern ist noch das Retten des Programmzeigers auf den Stack. Dies wird gemacht, um einen »syntax error« bei Aufruf des Beispielprogramms per SYS zu vermeiden. Wird das Beispiel durch einen SYS-Befehl aufgerufen, führt das »rts« der Beispielroutine zur Interpreter-Hauptschleife zurück. Das »rts« bezeichnet also sozusagen das Ende der Bearbeitung des BASIC-Befehls SYS. Die Hauptschleife prüft nun, ob der Programmzeiger auf einem Doppelpunkt oder einem Nullbyte steht, was gleichbedeutend damit ist, daß ein BASIC-Befehl ordnungsgemäß abgearbeitet werden konnte, ohne daß noch unzulässige Zeichen in der BASIC-Zeile übriggeblieben sind.
Dadurch, daß wir den Programmzeiger innerhalb des Beispiels auf vnam setzen, wird die Hauptschleife nach dem rts irgend etwas, nur kein BASIC-Trennzeichen finden, was logischerweise zu einem »syntax error« führt.

5.9 BASIC-Befehle, die Maschinenroutinen unterstützen

5.9.1 Der SYS-Befehl

Der wohl bekannteste BASIC-Befehl, der im direkten Zusammenhang mit der Benutzung eigener Maschinenprogramme steht, ist der Befehl SYS. Die Syntax des SYS-Befehls ist etwas anders als die beim C64:

```
sys adresse, AC, XR, YR, ST
```

Man kann also schon in der Parameterliste des SYS-Befehls die einzelnen Prozessorregister mit einem bestimmten Wert vorbesetzen. Diese Eingaben sind wie in manchen BASIC-7.0-Befehlen optional, das heißt es sind Parameterlisten der Art:

```
sys adresse,,, YR, ST
```

oder

```
sys adresse,, XR
```

gestattet. Fehlt die explizite Angabe eines Parameters zwischen zwei Kommas, entspricht der Wert dieses Registers nach dem Aufruf der Maschinenroutine dem des entsprechenden Übergabe-Bytes in der Zeropage. Die Übergabe-Bytes sind dabei dieselben, die JSRFAR benutzt, um Parameter zu übergeben.
Soweit, so gut, die Parameterübergabe beim SYS-Befehl kann aber noch weiter getrieben werden. Eine einfache Vorbelegung der Registerinhalte reicht ja insbesondere für längere Routinen nicht aus, um alle benötigten Parameter zu übergeben:
Dazu macht man sich zunutze, daß der eigentliche Sprung des SYS-Befehls zur eigenen Routine in dem Moment ausgeführt wird, wo der Interpreter, nachdem er die Sprungadresse gelesen hat, kein Komma mehr findet, das weitere Parameter anzeigt. Schreiben wir einmal eine solche Parameterliste, die eigene und vom Interpreter auszuwertende Parameter enthält:

```
sys adresse , AC ; a$,b$
```

oder

```
sys adresse ; 1234
```

In dem Moment, wo der Interpreter den Strichpunkt oder ein anderes Zeichen ungleich dem Komma liest, erfolgt der Sprung in unsere eigene Routine, der Programmzeiger TXTPTR steht innerhalb des BASIC-Textes genau auf unserem Semikolon oder sonstigem Trennzeichen. Die weitere Parameterauswertung kann daraufhin auf die inzwischen schon bekannte Weise mit den Routinen des Interpreters (CHRGET, CHKKOM, FRMEVL, FRESTR, DOSPAR usw. usf.) im eigenen Maschinenprogramm weitergeführt werden. Das einzige, auf das im Sinn einer geregelten Programmführung nach der Rückkehr aus der eigenen Routine geachtet werden muß, ist, daß TXTPTR nach Auswertung der Parameterliste auf einem BASIC-Trennzeichen (Nullbyte oder Doppelpunkt) zu stehen kommt. Ist dies nicht der Fall, gibt es einen »syntax error«.

5.9.2 Der Befehl RREG

Der RREG-Befehl ist gegenüber dem alten BASIC 2.0 des Commodore 64 ein ganz neuer Befehl. Er erlaubt die Übernahme der Prozessorregister nach Rückkehr von einem Unterprogrammaufruf in Variable. Diese Variablen müssen naturgemäß vom Typ her numerisch sein, Strings kennt der Prozessor ja nicht.

Die Syntax des RREG-Befehls ist ganz analog zu der Parameterliste des SYS-Befehls:

```
rreg AC, XR, YR, ST
```

oder

```
rreg ,, YR, ST
```

In diesem Fall sind die Parameter jedoch keine numerischen Ausdrücke, die in die Prozessorregister übernommen werden, sondern Variablennamen, beziehungsweise Namen von Feldelementen:

```
rreg a, x, y, s
```

oder

```
rreg r(0), r(1), r(2), r(3)
```

Die Übernahme der Werte in die Variablen erfolgt nicht direkt aus den Prozessorregistern, sondern wieder aus der Zeropage. In den bekannten Zeropagebytes liefert die JSRFAR-Routine bekanntlich die einzelnen Werte zurück.

5.9.3 Der POINTER-Befehl

Der POINTER-Befehl bietet eine etwas umständliche Möglichkeit, die Adresse einer Variablen an eigene Maschinenprogramme zu übergeben. Umständlich in dem Fall, daß einem der direkte Weg, der in vorigen Kapiteln beschrieben wurde, bekannt ist. Die Syntax:

```
var1 = pointer (var2)
```

Als Ergebnis dieser Zuweisung erhielte man in der Variablen var1 als Fließkommazahl die Adresse der Variablen var2 in Bank 1. Innerhalb der SYS-Parameterliste könnte man den POINTER-Befehl zum Beispiel so anwenden:

```
sys adresse, pointer(var) and 255, int (pointer(var)/256)
```

Man erhielte dann im Akkumulator das Lowbyte der Adresse der Variablen, im X-Register das Highbyte. Als Variable sind auch Feldelemente zugelassen, da die Felder aber beim Anlegen neuer nichtdimensionierter Variabler verschoben werden, sollte in diesem Fall die Definition aller nichtdimensionierten Variablen vor dem Einsatz des POINTER-Befehls erfolgen. Dieses Problem hat man bei Einsatz der Interpreter-Routinen FETVAR beziehungsweise GEVTAR natürlich nicht.

5.9.4 Der Befehl USR (...)

Wie bei den im nächsten Kapitel besprochenen BASIC-Vektoren, ist auch an dieser Stelle eine Unterscheidung zwischen dem einfachen Aufruf von Maschinenprogrammen (SYS) und der Einbindung von Assemblerroutinen in einen Ausdruck, ob numerisch oder

String, zu unterscheiden. Diese Einbindung erfolgt mit dem USR-Befehl. Der Befehl erhält als Eingangsparameter den Wert eines Arguments, das dem USR-Kommando in Klammern eingeschlossen folgt. Ein Aufruf könnte beispielsweise sein:

```
var = 1234 * usr (1.24)
```

Um die Adresse des eigenen Maschinenprogramms mit dem USR-Befehl zu verbinden, wird die Adresse in der Reihenfolge Low-/Highbyte in die beiden Speicherzellen:

usrvec = $1219/ $121a

geschrieben. Das Ergebnis der Rechnung in der eigenen Routine wird wieder in FAC übergeben.

5.10 BASIC im Interrupt

Obwohl Grafik und Sound nicht eigentlich Gegenstand dieses Buches sein sollen, kommt man nicht darum herum, diese zumindest am Rande zu streifen.
BASIC besitzt nämlich eine eigene Interruptroutine, innerhalb derer VIC II und SID beeinflußt werden. Die BASIC-Interruptroutine wird regelmäßig während des Tastaturinterrupts angelaufen. Ein Flag in der erweiterten Zeropage bestimmt, ob die BASIC-Interruptroutine bearbeitet wird oder nicht:

SPRINT = $12fd

Dieses Flag spielt genau die gleiche Rolle, die »flag« im Beispielprogramm 4.1.2.1 eingenommen hat. Ein Wert ungleich Null in SPRINT bedeutet, daß die Interruptroutine gerade in Bearbeitung ist. Das bewirkt, daß die Routine für weitere Aufrufe vorübergehend gesperrt ist. Durch Beschreiben von SPRINT mit einem Wert ungleich Null kann man auf die einfachste Art den BASIC-Interrupt ausschalten!
Daß ein solches Flag notwendig ist, sieht man aus den umfangreichen Arbeiten, die BASIC innerhalb des Interrupts zu erledigen hat. Im einzelnen müssen nämlich bearbeitet werden:

Sprite-Bewegungen für 8 Sprites
Sprite-Kollisionserkennung
Abfrage eines Lightpens
Tonerzeugung

Diese vier Punkte sollen einzeln behandelt werden.

5.10.1 Sprite-Bewegungen

Um ein Sprite auf dem Bildschirm darstellen zu können, benötigt der VICII verschiedene Angaben in seinen Sprite-Registern. Innerhalb des Interrupts werden nur die Register des VIC verändert, die die Sprite-Positionen festlegen. Durch eine Positionsveränderung von Interrupt zu Interrupt wird die Sprite-Bewegung erreicht.
Die aktuellen Sprite-Positionen werden festgehalten in der erweiterten Zeropage bei:

SPRPOS = $11d6 bis $11e6

Hier liegen die einzelnen Angaben in der gleichen Reihenfolge, wie sie auch in den VIC-Registern vorliegen. Also:

$11d6	X-Koordinate Sprite 0, Bits 0 bis 7
$11d7	Y-Koordinate Sprite 0
$11d8	X-Koordinate Sprite 1, Bits 0 bis 7
$11d9	Y-Koordinate Sprite 1
.	
.	
.	
$11e4	X-Koordinate Sprite 7, Bits 0 bis 7
$11e5	Y-Koordinate Sprite 7
$11e6	Höchstwertige Bits der X-Koordinaten (Bit 0 gehört zu Sprite 0, Bit 1 zu Sprite 1 usw.)

Diese insgesamt 17 Speicherzellen werden bei jedem Interrupt in die entsprechenden VIC-Register ab $d000 geschrieben, natürlich nur dann, wenn der Durchlauf durch die Interruptroutine nicht durch SPRINT verhindert wird.
Versucht man, diese VIC-Register auf direktem Weg zu beschreiben, ohne daß die BASIC-Interruptroutine ausgeschaltet ist, wird man nicht viel Erfolg haben. Den Grund kennen wir jetzt.
Das Sprite-Enable-Register $d015 wird während des Interrupts nicht verändert. Um Sprites ein- und auszuschalten, ist es deshalb notwendig, dieses Register direkt zu beschreiben.
Probieren Sie zunächst doch einmal mit dem Monitor aus, welche Wirkung die Speicherzellen SPRPOS haben. Belegen Sie aber vorher das Register $d015 mit $ff, um alle Sprites einzuschalten.

Es kommt aber noch besser: Nicht nur die Positionierung der Sprites erfolgt während des Interrupts, sondern auch eine Sprite-Bewegung, die sich von Maschinensprache aus gut

beeinflussen läßt. Da alle Bewegungen während des Interrupts erfolgen, genügt es, einige Werte in die zugehörigen Speicherzellen in der erweiterten Zeropage zu schreiben, um zu erreichen, daß die Sprite-Bewegung sodann sozusagen vollautomatisch erfolgt.

Zunächst einmal zur Darstellungsweise einer Sprite-Bewegung:

Die Bewegung läßt sich grundsätzlich zerlegen in zwei Anteile: eine Bewegung in X- und eine Bewegung in Y-Richtung (horizontale und vertikale Bewegung).
Jede der beiden Bewegungen parallel zu den Koordinatenachsen zerfällt zudem in zwei weitere Anteile: den Betrag und die Richtung der Geschwindigkeit.
Jede dieser insgesamt vier Angaben muß wiederum für alle acht Sprites in irgendeiner Form festgehalten werden, damit die Bewegungen vollständig beschrieben sind.
Die Angaben befinden sich wieder in der erweiterten Zeropage, wo sonst. Zu jedem Sprite gehört ein Feld von 11 Byte:

Sprite-Nummer	Feld
0	$117e bis $1188
1	$1189 bis $1193
2	$1194 bis $119e
3	$119f bis $11a9
4	$11aa bis $11b4
5	$11b5 bis $11bf
6	$11c0 bis $11ca
7	$11cb bis $11d5

Das gesamte Feld hat auch einen Namen:

SPRDAT = $117e

Betrachten wir jetzt ein einzelnes 11-Byte-Feld eines Sprites:

Byte 0:
Anzahl der Bewegungsschritte pro Durchlauf durch die Interruptroutine. Dies entspricht einem Anteil der Geschwindigkeit des Sprites.

Byte 1:
Temporärer Zähler für die obige Angabe. Das Beschreiben dieser Speicherzelle bringt nichts.

Byte 2:
Zusammengesetzte Richtung der Sprite-Bewegung. Die Richtungen der Bewegung werden nicht einzeln für die X- und die Y-Bewegung aufbewahrt, sondern in Form eines Bytes,

das auch manchmal Quadrant genannt wird, was aber nicht hundertprozentig zutrifft. Eine Sprite-Bewegung vorausgesetzt, die schräg über den Bildschirm verläuft, die also sowohl einen X- als auch einen Y-Anteil besitzt, hat der Quadrant folgende Bedeutung:

Quadrant	Richtung
1	von links oben nach rechts unten
2	von rechts oben nach links unten
3	von rechts unten nach links oben
4	von links unten nach rechts oben

Andere Werte des Quadranten haben zyklisch die gleiche Bedeutung: 5 entspricht 1, 6 entspricht 2, usw.

Bytes 3 + 4:

Schrittweite der Sprite-Bewegung in X-Richtung in der Reihenfolge Low-/Highbyte. Während Byte 0 die Anzahl der Schrittweiten-Additionen pro Interrupt festlegt, wird hier die Größe der einzelnen Schritte bestimmt. Zum Maßstab der Schrittweite sowie überhaupt zu den noch folgenden Koordinatenangaben muß noch ein Wort gesagt werden:
Alle diese Angaben erfolgen nicht im Sprite-Koordinatensystem, sondern in einem System, das eine feinere Auflösung zuläßt. Dies wird notwendig, wenn man die Geschwindigkeit bedenkt, mit der die einzelnen Interrupts aufeinanderfolgen. Würde ein Sprite pro Interrupt nur um einen Rasterpunkt bewegt, bedeutete dies schon eine Bewegung um 60 Punkte pro Sekunde. Dieser Maßstab ist also nicht fein genug.
Am besten stellt man sich die Schrittweitenangaben und die noch folgenden Koordinatenangaben als Nachkommastellen vor. Erst wenn eine Addition der Nachkomma-Schrittweite kleiner als ein Rasterpunkt zu den Nachkomma-Koordinaten einen Überlauf ergibt, wird die tatsächliche Sprite-Position in SPRPOS um eins erhöht beziehungsweise vermindert.

Bytes 5 + 6:

Schrittweite in Y-Richtung in Low- und Highbyte.
Wie man sich leicht vorstellen kann, ergibt sich aus dem Verhältnis der beiden Schrittweiten der Winkel der Sprite-Bewegung. Es gilt:

```
abs(tan(winkel)) = dy/ dx
```

wobei dy und dx die beiden Schrittweiten darstellen, »winkel« ist der Winkel des Geschwindigkeitsvektors mit der positiven X-Achse.

Bytes 7 + 8:

»Nachkomma«-Koordinaten in X-Richtung. Die beiden Bytes werden zu Rechenzwecken benutzt, deswegen ergibt ihre Änderung keinen Effekt.

Bytes 9 + 10:
»Nachkomma«-Koordinaten in Y-Richtung.

Unsere Erkenntnisse können wir natürlich auch einmal ausprobieren:

Beschreiben Sie zunächst das Sprite-Enable-Register ($d015) mit dem Wert $ff, was alle Sprites einschaltet. Dies geschieht am besten vom Monitor aus durch den Befehl

m fd015

und Überschreiben des ersten angezeigten Bytes mit »ff«.

Zweiter Schritt: Wir positionieren das Sprite mit Nummer 0 auf X = $50 und Y = $50. Geben Sie zu diesem Zweck den Befehl

m 11d6

ein und beschreiben Sie die ersten beiden Bytes mit »50«.

Letzter Schritt: Wir setzen das Ganze in Bewegung. Hierfür lassen wir uns mit

m 117e

das zu Sprite 0 gehörige SPRDAT-Feld anzeigen.
Beschreiben Sie jetzt zuerst das erste Byte mit der Schrittzahl pro Interrupt, beispielsweise $10. Noch tut sich nichts, denn die Schrittweiten in beiden Richtungen werden in der Regel noch Null sein. Beschreiben Sie auch diese Bytes, beginnt sich das Sprite zu bewegen. Wählen Sie die beiden Schrittweiten gleich groß, erhalten Sie einen Winkel von 45 Grad für die Bewegung.
Durch Beschreiben des Quadranten-Bytes erreichen Sie, daß die Richtung der Bewegung festgelegt wird.

5.10.2 Sprite-Kollisionen

Um auftretende Kollisionen zwischen Sprites beziehungsweise auch Kollisionen von Sprites mit Hintergrundzeichen zu erfassen, fragt die BASIC-Interruptroutine das relevante VIC-Register

IRR = $d019

ab. Von Bedeutung sind für Kollisionen die Bits 1 und 2, die einmal eine Interruptanforderung durch die Kollision eines Sprites mit dem Hintergrund, zum anderen eine Kollision zweier Sprites anzeigen.
Die Wirkung der BASIC-Interruptroutine ist in diesem Fall nicht sofort sichtbar wie bei den Sprite-Bewegungen. Es werden vielmehr einige Flags gesetzt, die von den zuständigen BASIC-Routinen abgefragt werden können. Die Benutzung dieser BASIC-Flags ist aber auch für den Maschinenprogrammierer bequemer, als eine eigene Routine zur Abfrage der VIC-Register zu schreiben.

Zunächst liegen die Sprite-Kollisionsregister in der erweiterten Zeropage:

COLLSP = $11e7	entspricht dem VIC-Register 30. Jedes gesetzte Bit entspricht einem an einer Kollision Sprite/Sprite beteiligten Sprite
COLLBG = $11e8	entspricht dem VIC-Register 31 für Kollisionen eines Sprites mit einem Hintergrundzeichen

Daß sich in diesen beiden Bytes Informationen über kollidierte Sprites befinden, wird in zwei Flags angezeigt:

CSPFLG = $1276	Flag: Kollision Sprite/Sprite hat stattgefunden ($ff)
CBGFLG = $1277	Flag: Kollision eines Sprite mit dem Hintergrund ist eingetreten ($ff)

Noch ein weiteres Flag ist in diesem Zusammenhang zu nennen:

WORKIN = $127f	Flag für zu bearbeitenden Kollisionstyp. Ist Bit 0 gesetzt, wird die Kollision Sprite/Sprite bearbeitet und die obigen Flags entsprechend gesetzt. Bit 1 ist zuständig für die Bearbeitung der Kollision Sprite/Hintergrund. Diese Bits werden unter BASIC vom Befehl COLLISION gesetzt. Mit Bit 7 kann eine Kollisionsbearbeitung unterbunden werden. Das gesetzte Bit zeigt in der nun schon bekannten Weise an, daß eine Kollisionsbearbeitung aktuell gerade im Gange ist. Die entsprechenden Routinen werden dann nicht durchlaufen.

5.10.3 Lichtgriffel (Lightpen)

Die Behandlung eines Lightpens während des Interrupts ist eng mit der vorhergehenden Sprite-Kollisions-Behandlung verbunden. Dies geht schon daraus hervor, daß der Lightpen-Interrupt im gleichen VIC-Register angezeigt wird wie die Kollisionen (Bit 3, $d019).
Auch die Tatsache, daß eine Lightpen-Abfrage angesagt ist, wird an der gleichen Stelle wie bei den Sprite-Kollisionen hinterlegt:

WORKIN, Bit 2	Flag: Lightpen bearbeiten

Wird während der BASIC-Interruptroutine festgestellt, daß der Interrupt durch den Lichtgriffel erfolgt ist, und ist die Bearbeitung des Lightpens durch Bit 2 von WORKIN erlaubt, wird das Flag

LIGHTP = $1278

gesetzt ($ff). Die Position des Lightpen wird aus den VIC-Registern $d013 und $d014 in die erweiterte Zeropage übernommen:

XPEN = $11e9	horizontale Lightpen-Position
YPEN = $11ea	vertikale Position

5.10.4 SID im Interrupt

Wie schon bei der Behandlung des VIC, wird auch im Zusammenhang mit der Programmierung des SID der Interrupt dazu benutzt, zeitliche Abläufe zu programmieren. War es beim VIC die Bewegung von Sprites, so handelt es sich im Fall des SID um zwei Arten von zeitlich zu steuernden Vorgängen:

- Die Erzeugung von Tönen einer bestimmten Dauer
- Die Steuerung des Frequenzverlaufes von Tönen in Abhängigkeit von der Zeit.

Zur Erinnerung: Die Amplitudenmodulation, der zeitliche Verlauf der Lautstärke, kann direkt durch den SID erfolgen (ATTACK, DELAY ...).
Da die durch die BASIC-Interruptroutine zur Frequenz- und Tondauersteuerung benutzten Speicherzellen nur schwierig zu beeinflussen sind, sollen sie hier nur am Rande erwähnt werden. Zur Übertragung von Steuerwerten in die Interrupt-Speicher geht man besser denselben Weg wie der Interpreter: Er benutzt einen Parameterbereich, der durch eine spezielle Routine in die Interruptspeicher kopiert wird.
Diesen Parameterbereich wollen wir uns ein wenig näher ansehen:

TMPTIM = $12a3/ $12a4	Tondauer in der Reihenfolge Low-/Highbyte
MAXFRQ = $12a5/ $12a6	Maximalfrequenz bei oszillierendem Frequenzverlauf
MINFRQ = $12a7/ $12a8	Minimalfrequenz bei oszillierendem Frequenzverlauf
DIRECT = $12a9 #	Richtung des Frequenzverlaufs (0 = auf, 1 = ab, 2 = oszillierend)
STEP = $12aa/ $12ab	Schrittweite bei Frequenzänderungen

FREQU = $12ac/ $12ad	Tonfrequenz
PULSE = $12ae/ $12af	Tastverhältnis bei Rechteckschwingungen
WAVE = $12b0	Schwingungsform ($11 = Dreieck, $21 = Sägezahn, $41 = Rechteck, $81 = rosa Rauschen, Kombinationen sind erlaubt). Der Inhalt dieses Registers entspricht den Kontrollregistern des SID.

Hinzu kommt eine Speicherzelle etwas außerhalb:

VOICE = $1281	Betroffene Stimme (0 bis 2)

Nachdem die Register des Parameterbereiches entsprechend belegt wurden, können sie mit der Interpreter-Routine

TONE = $726b

in die Interrupt-Speicherzellen übertragen werden. Dabei wird gewartet, bis die letzte Tonausgabe beendet ist! TONE erhält keine Parameter in den Prozessorregistern und liefert auch keine zurück.

Das Ganze an einem Beispiel:

Beschreiben Sie mit Hilfe des Monitors die Parameterzellen mit folgenden Werten:

```
012a3   20 00 00 60 00 70 02 10
012ab   00 00 60 00 00 11 xx xx
```

Die Stimme wird gesetzt durch

```
01281   01 xx xx xx xx xx xx xx
```

Starten Sie jetzt die Parameterübergabe mit dem Befehl

```
j f726b
```

Sie hören einen trillernden Ton (Frequenzverlauf oszillierend). Durch Überschreiben des Parameterbereiches mit neuen Werten und einem Neustart der Übertragungsroutine können Sie die verschiedenen Möglichkeiten ausprobieren.

6 BASIC-Vektoren in der erweiterten Zeropage

Genau wie das Betriebssystem besitzt auch der Interpreter verschiedene Vektoren in der erweiterten Zeropage. Diese Vektoren ermöglichen einen bequemen Eingriff in die wichtigsten Routinen des Interpreters.
Der User-Vektor, der Vektor, der zum BASIC-Befehl USR gehört, ist im vorangegangenen Kapitel 5.9.4 zu finden.

6.1 Usertoken

Drei der BASIC-Vektoren sind auch für den schon am 64er Geübten ganz neu. Sie erlauben das Einschleusen neuer BASIC-Befehle in den Interpreter. Die Befehlsworte werden dabei wie die Schlüsselworte sonst auch in Token übersetzt. Beim Listen verläuft die Umwandlung in der anderen Richtung, die Token werden wieder in Befehlsworte verwandelt.
Die zuständigen Vektoren:

IESCFN = $2fc	Zum Einschleusen neuer Funktionen ($4c78)
IESCLK = $30c	Übersetzung neuer Schlüsselworte in Token ($4321)
IESCPR = $30e	Listen neuer Token ($51cd)
IESCEX = $310	Neuen Befehl ausführen ($4ba9)

Usertoken bestehen immer aus 2 Byte. Das erste Token-Byte, das man auch Token-Präfix nennen könnte, kann zwei Werte annehmen:

Ist es gleich $fe, so muß das zweite Token gleich 1 oder größer als 38 = $26 sein. Die Usertoken mit dem Präfix $fe sind zur Einbindung von normalen Befehlen vorgesehen, wie es zum Beispiel ein neuer Befehl »merge« darstellen würde.
Ist das Token-Präfix gleich $ce, darf das zweite Token-Byte gleich 1 oder größer als 10 = $0a sein. Diese Usertoken werden benutzt, um neue Funktionen in die Arithmetik einzuführen. Die Funktionen werden immer mit einem Argument in runden Klammern erwartet.
Für beide Fälle folgen noch Beispiele, wenn es noch nicht klar sein sollte.

6.1.1 Eigene Key-Worte in Token übersetzen (IESCLK = $30c)

Der erste Schritt bei der Einführung eigener BASIC-Befehle über Usertoken besteht darin, dafür zu sorgen, daß die eigenen Befehlsworte in geeignete Token umgewandelt werden. Dabei ist es gleichgültig, ob es sich bei den eigenen Token um Befehls-Usertoken oder um Funktions-Usertoken handelt.
Die Erzeugung der Usertoken aus den Schlüsselworten wird wesentlich dadurch vereinfacht, daß der Interpreter eine Routine bereitstellt, die Schlüsselworte innerhalb einer geeignet aufgebauten Tabelle erkennen kann:

SEARCH = $43e2

Die SEARCH-Routine benötigt bei ihrem Aufruf lediglich zwei Parameter:

A = Highbyte der Tabellenadresse der eigenen Key-Worte
Y = Lowbyte der Tabellenadresse

Bitte beachten Sie, daß die Übergabe der Tabellenadresse nicht in der üblichen Reihenfolge Lowbyte im Akkumulator vonstatten geht!
Das Ergebnis der Suche erscheint im Statusbyte des Prozessors, im Akkumulator und in der Zeropage-Adresse $0d:

Minus-Flag gesetzt:	Wort gefunden
Falls gefunden:	A = Nummer des Wortes in der Tabelle geodert mit $80
	dito in COUNT = $0d in der Zeropage

Wie schon erwähnt, muß die Tabelle der Key-Worte einen bestimmten Aufbau haben, damit SEARCH eingesetzt werden kann. Dazu werden die einzelnen Schlüsselworte hintereinander abgelegt, wobei als letzter Buchstabe des Key-Worts ein Großbuchstabe verwendet wird. Das Ende der Tabelle wird durch ein Nullbyte gekennzeichnet.

Beispiel:

.byte "mergEolDdumPquaD",0

Soweit zur SEARCH-Routine. Wie aber setzt man diese Routine ein, damit ein eigenes Key-Wort in Token umgewandelt werden kann?

Der Vektor IESCLK liegt innerhalb der Routine, die jede BASIC-Zeile auf BASIC-Schlüsselworte absucht und diese in Token umwandelt. Der Sprung über den IESCLK-Vektor liegt dabei ganz am Anfang dieser Routine, bevor eine Prüfung auf die normalen BASIC-Befehlsworte stattgefunden hat.
Normalerweise erfolgt der Sprung über diesen Vektor mit gesetztem Carry-Flag, was bewirkt, daß die normale BASIC-Tokenerzeugung erfolgt. Verbiegen wir den IESCLK-Vektor auf eine eigene Routine, dient das gesetzte Carry als Flag dafür, daß an dieser Stelle kein eigenes Schlüsselwort erkannt wurde.
Ist das Carry-Flag gelöscht, gilt dies als Anzeichen dafür, daß eine User-Befehlserkennung erfolgreich abgeschlossen wurde. In diesem Fall wird das zweite Tokenbyte im Akku erwartet. Im X-Register wird die Information übergeben, welches Token-Präfix ausgewählt wurde:

X = $ff:	Das Präfix soll $ce sein
sonst:	Präfix $fe

Um das Einfügen der beiden Tokenbytes in die übersetzte BASIC-Zeile braucht man sich nicht mehr selbst zu kümmern. Dies erfolgt dann automatisch.
Probieren wir das Ganze einmal an einem Beispiel durch, um zu sehen, wie es funktioniert: Als neue Befehlsworte sollen »merge« und »old« dienen. Die zugehörigen Token sollen ($fe, $27) und ($fe, $28) sein:

```
        .base $1300

 .define iesclk = $30c          ; Vektor: Umwandlung in Token
 .define oldilk = $4321         ; alter Vektorinhalt
 .define search = $43e2         ; Befehlsworterkennung

 start       lda #<(own)        ; Vektor auf eigene Routine
             ldx #>(own)        ; verbiegen
             jmp m1

 ende        lda #<(oldilk)     ; alten Vektor
             ldx #>(oldilk)     ; restaurieren
 m1          sta iesclk
```

```
        stx iesclk+1
        rts

own     tax                 ; letztes Zeichen retten
        lda #>(tab)         ; Startadresse der Tabelle an
        ldy #<(tab)         ; SEARCH übergeben
        jsr search          ; Key-Wort suchen
        bpl nfound          ; Plus: nicht in Tabelle
        and #%0111'1111     ; drin: Bit 7 der Nummer löschen
        clc                 ; Carry clear für Addition
        adc #$27            ; Nummer in Tabelle + $27
        ldx #0              ; Flag für Präfix $fe
        beq back            ; Carry ist gelöscht!

nfound  sec                 ; Flag für nicht gefunden
        txa                 ; Akku zurückholen
back    jmp oldilk

tab     .byte "mergEolD",0  ; Tabelle der Key-Worte
```

Ich denke, der Aufbau der Beispielroutine dürfte soweit klar sein. Das X-Register wird benutzt, um den Akkumulatorinhalt zu retten, da das Register durch die SEARCH-Routine nicht benutzt wird. Das Retten des Akkumulators ist notwendig, damit die Umwandlungsroutine im Fall, daß kein User-Key-Wort gefunden wurde, richtig weiterarbeiten kann: Der Akku enthält das zuletzt aus der BASIC-Zeile gelesene Zeichen.

6.1.2 Listen eigener Token (IESCPR = $30e)

Schritt 2 beim Einbau eigener Befehle ist folgerichtig, daß eigene Token beim Listen auch wieder in die entsprechenden Befehlsworte umgewandelt werden. Der Hebelpunkt für diesen Zweck ist der Vektor IESCPR.

IESCPR liegt innerhalb der LIST-Routine. Wird der Vektor auf eine eigene Routine umgeschaltet, erhält man beim Eintritt in das eigene Programm die folgenden Parameter in den Prozessorregistern:

X =	$ff, wenn das Usertoken das Präfix $ce besitzt 0, wenn das Präfix gleich $fe ist
A =	2. Tokenbyte

In Abhängigkeit dieser zwei Werte muß unser eigenes List-Programm das entsprechende Befehlswort ausgeben. Besser gesagt: Es genügt auch, einen Zeiger auf das auszugebende Befehlswort in

INDEX = $24/ $25

an den Interpreter zu übergeben, wobei das gelöschte Carry-Flag anzeigt, daß der Interpreter das Befehlswort ausgeben soll. INDEX muß zu diesem Zweck auf das erste Zeichen des Befehlswortes weisen.
Diese Methode ist aber unnötig umständlich. Eine andere ROM-Routine erledigt auch die Suche des Key-Wortes noch für uns:

USTLST = $516a

Die Routine benötigt für ihre Arbeit:

im X-Register	die Nummer des Key-Worts in unserer Tabelle der Befehlsworte geodert mit $80
im Akkumulator	das Highbyte der Tabellenadresse
im Y-Register	das Lowbyte der Tabelle

Das Carry wird bei Einsatz dieser ROM-Routine immer gesetzt, da sonst auch der Interpreter noch versucht, etwas auszugeben, was nicht in unserem Sinne ist.

Wieder ein Beispiel, das auf das vorige Bezug nimmt:

```
        .base $1300

 .define iescpr = $30e        ; Vektor: Umwandlung Token in Text
 .define oldipr = $51cd       ; alte Vektoradresse
 .define ustlst = $516a       ; Listen eines Tokens

 start      lda #<(own)       ; Vektor auf eigene
            ldx #>(own)       ; Routine
            jmp m1

 ende       lda #<(oldipr)    ; alten Vektorinhalt
            ldx #>(oldipr)    ; wiederherstellen
 m1         sta iescipr
            stx iescipr+1
            rts
```

```
own         cpx #$ff           ; Flag für Präfix $ce gesetzt?
            beq not            ; ja: kein Listen
            cmp #$27           ; 2. Tokenbyte kleiner $27?
            bcc not            ; ja: nicht Listen
            cmp #$29           ; 2. Byte größer als $28
            bcs not            ; ja: nicht Listen
            sbc #$26           ; sonst: $27 subtr. (Carry=1!)
            ora #$80           ; 128 addieren
            tax                ; in X übergeben
            lda #>(tab)        ; Startadresse der Tabelle
            ldy #<(tab)        ; übergeben
            jsr ustlst         ; Befehlswort listen
not         sec                ; Flag: keine Ausg. durch Interpr.
            jmp oldipr         ; zurück zur LIST-Routine

tab         .byte "mergEolD",0 ; Tabelle der Befehle
```

Die Wirkung der Routine läßt sich mit Hilfe des Monitors leicht einmal nachprüfen, indem die erforderlichen Token ($fe,$27) für merge oder ($fe,$28) für old per Hand in eine BASIC-Zeile eingefügt werden.

6.1.3 Ausführen eigener Maschinenprogramme

Nachdem das Eingeben und Listen eigener Befehlsworte schon ganz gut funktioniert, fehlt nur noch, die den Befehlsworten zugeordneten Maschinenroutinen ausführen zu lassen. Dies wird durch die beiden Vektoren IESCFN = $2fc für eigene Funktionen und IESCEX = $310 für eigene Befehle möglich.

6.1.3.1 Ausführen eigener Funktionen (IESCFN)

Der Vektor zur Ausführung von Funktionen, die durch Usertoken in das BASIC eingebunden sind, liegt innerhalb der Unterroutine EVAL von FRMEVL, die einzelne Terme nach FAC beziehungsweise in die entsprechenden String-Register holt.
Einzelne Terme eines Ausdrucks können sein:

- Zahlenkonstante
- Variable
- Ausdrucke in Klammern
- Funktionen
- Strings in Anführungszeichen

In die Erkennung der Funktionen – normalerweise zum Beispiel SIN(...), SQR(...) oder auch MID$(...), LEFT$(...) – kann man über den IESCFN-Vektor eingreifen.

Token mit Token-Präfix $ce und zweitem Tokenbyte gleich 1 oder größer als 10 führen zu einem Sprung mit »jsr« über den IESCFN-Vektor.

Bei diesem Unterprogrammaufruf befindet sich das 2. Tokenbyte im Akkumulator, so daß in einem eigenen Erkennungsprogramm zu unterschiedlichen Funktionsberechnungen verzweigt werden kann. Das Funktionsargument in Klammern ist bereits ausgewertet und befindet sich im FAC beziehungsweise in den String-Zeigern. Die Prüfung, ob auf das Argument »Klammer zu« folgt, ist dem Anwender vorbehalten. Dadurch ist es denkbar, Funktionen einzuführen, die mit mehreren, durch Komma voneinander getrennten Argumenten arbeiten. Man denke zum Beispiel an die Vektor- oder Matrizenrechnung.

Die einzelnen Maschinenprogramme, die die User-Funktion berechnen, müssen mit »rts« abgeschlossen sein. Dieses »rts« führt zur EVAL-Routine zurück.
Ist das Carry beim rts gesetzt, gilt dies als Anzeichen dafür, daß bei der Funktionsbearbeitung ein Fehler gefunden wurde. Ist es gelöscht, wird das Funktionsergebnis auf den Werttyp numerisch geprüft!

Diesen kleinen Fallstrick kann man umgehen, indem man die erste RTS-Adresse vor dem Rücksprung zum Interpreter entfernt. Dadurch gelangt man direkt zur Routine FRMEVL, von der EVAL aufgerufen wurde.

Als Beispiel soll an dieser Stelle eine Funktion ASN(...) eingebaut werden, die den Arcussinus eines Argumentes zwischen -1 und +1 berechnet. Der Einfachheit halber wird dazu die Formel aus dem C128-Handbuch benutzt:

arcsin(x) = arctan(x/(1-x*x)↑ 0.5)

Eine Reihenentwicklung des Arcussinus wäre zwar sicher genauer, als Beispiel reicht diese Formel aber vollkommen aus.

```
        .base $1300

    .define iesclk  = $30c      ; Vektor: Token bilden
    .define iescpr  = $30e      ; Vektor: Token listen
    .define iescfn  = $2fc      ; Vektor: Funktion ausführen
    .define oldilk  = $4321     ; alter Vektor IESCLK
    .define search  = $43e2     ; Suchen in Befehlstabelle
    .define ustlst  = $516a     ; Listen von Token
    .define oldipr  = $51cd     ; alter Vektor IESCPR
    .define mularg  = $8a2c     ; FAC := FAC * ARG
```

```
.define inttofk = $af03      ; (A,Y) in Fließkomma wandeln
.define subarg  = $af15      ; FAC := ARG - FAC
.define divarg  = $af27      ; FAC := ARG / FAC
.define sqrfac  = $af30      ; Quadratwurzel
.define chssgn  = $af33      ; FAC := - FAC
.define atgfac  = $af48      ; Arcustangens
.define movcona = $af5d      ; ((A,Y)) nach ARG holen
.define chksgn  = $795e      ; Zeichen vergleichen
.define stofac  = $af66      ; FAC nach ((X,Y)) bringen
.define factoa  = $af6c      ; ARG := FAC

start       lda #<(ownpr)    ; List-Vektor auf eigene Routine
            ldx #>(ownpr)
            sta iescpr
            stx iescpr+1
            lda #<(ownlk)    ; Token-Bildung auf eigene Routine
            ldx #>(ownlk)
            sta iesclk
            stx iesclk+1
            lda #<(exec)     ; Funktionsauswertung auf eigene
            ldx #>(exec)     ; richten
            sta iescfn
            stx iescfn+1
            rts

; Tabelle der Befehlsworte (nur ein Eintrag)

tab         .byte "asN",0

; Tokenbildung ASN = ($ce,1)

ownlk        tax             ; Zeichen in A retten
             lda #>(tab)     ; Highbyte Tabellenstart
             ldy #<(tab)     ; und Lowbyte übergeben
             jsr search      ; Suche in Tabelle
             bpl ol1         ; nicht drin: Sprung
             and #$7f        ; drin: Bit 7 clear
             clc
             adc #1          ; 1 addieren um 2. Byte zu erhalten
             ldx #$ff        ; Flag für Präfix $ce
             bne ol2         ; immer
```

```
ol1        txa              ; Akku wiederherstellen
           sec              ; Flag: nicht gefunden
ol2        jmp oldilk       ; zur alten Vektoradresse

; Listen des Usertokens ($ce,1)

ownpr      cpx #$ff         ; Flag für Präfix $ce gesetzt?
           bne op1          ; nein: Sprung
           cmp #1           ; 2. Byte gleich 1?
           bne op1          ; nein: Sprung
           sec              ; 1 subtrahieren, um Nummer in
           sbc #1           ; der Tabelle zu erhalten
           ora #$80         ; 128 addieren
           tax              ; Nummer in X übergeben
           lda #>(tab)      ; Tabellenadresse
           ldy <(tab)       ; übergeben
           jsr ustlst       ; Token listen
op1        sec              ; Flag: Interpr. nicht listen
           jmp oldipr       ; zur List-Routine

; Funktion ausführen (Definition siehe oben). FAC enthält
; bereits das Argument, das im folgenden mit W abgekürzt wird

exec       lda #")"         ; Prüfung auf Klammer zu
           jsr chksgn
           ldx #<(fsav1);   W in Sicherungsspeicher fsav1
           ldy #>(fsav1);   bringen
           jsr stofac
           jsr factoa       ; W nach ARG kopieren
           jsr mularg       ; W * W berechnen
           jsr factoa       ; FAC runden und nach ARG
           lda #0           ; Konstante 1 Highbyte
           ldy #1           ; Lowbyte
           jsr inttofk      ; in Fließkommazahl wandeln
           jsr subarg       ; FAC = W * W - 1
           jsr chssgn       ; Vorzeichen umdrehen
           jsr sqrfac       ; Wurzel ziehen
           lda #<(fsav1);   W aus dem Sicherungsspeicher
           ldy #>(fsav1);   in den ARG holen
           jsr movcona
           jsr divarg       ; dividieren
```

```
            jsr atgfac      ; Arcustangens bilden
            pla             ; RTS-Adresse vom Stack nehmen
            pla
            rts             ; fertig

fsav1       .space of 5     ; 5 Byte Sicherungsspeicher
```

Der Fall, daß der Arcussinus von 1 oder -1 berechnet werden soll – für diesen Wert ist unsere Berechnungsformel für den Arcussinus nicht definiert – wird übrigens von der Routine DIVARG abgefangen, die einen Error »division by zero« meldet. Bei gewissenhafter Programmierung müßten diese beiden Fälle natürlich gesondert behandelt werden.

6.1.3.2 Ausführen von Befehlen (IESCEX)

Der zu IESCFN entsprechende Vektor für die Ausführung neuer BASIC-Befehle, der Vektor IESCEX, liegt innerhalb der Interpreter-Hauptschleife.

Alle Befehlsaufrufe der Usertoken mit dem Präfix $fe und zulässigem zweiten Usertokenbyte (1 oder größer 38) werden über den IESCEX-Vektor geführt. Usertoken mit dem Präfix $ce werden von diesem Vektor also nicht erfaßt, ebensowenig wie die Token mit dem Präfix $fe über den Vektor IESCFN liefen.

Der Sprung über den IESCEX-Vektor erfolgt im Gegensatz zum IESCFN-Vektor nicht über einen Unterprogrammaufruf! Ein einfaches rts genügt also nicht, um in die Interpreter-Hauptschleife hinter dem Vektor zurückzukehren. Dazu benutzt man statt dessen einen Sprung zum ursprünglichen Inhalt des Vektors.

Diese Art der Rückkehr ist einmal dann von Interesse, wenn das festgestellte Usertoken nicht in unserer aktuellen Befehlsliste vorkommt: Ein Sprung mit gesetztem Carry an die alte Adresse des Vektors führt zur Ausgabe eines »syntax error«.

Zum anderen kann man durch den Rücksprung auch eine eigene Befehlsroutine aufrufen: Ein gelöschtes Carry beim Sprung zurück in die Interpreter-Hauptschleife führt zu einem direkten Sprung zur CHRGET-Routine, deren rts dann zu einer eigenen Befehlsroutine führen kann. Dazu wird in der eigenen Befehlserkennung die Adresse der Befehlsroutine (um eins vermindert) auf den Stack gelegt, wie es auch in dem folgenden Beispiel gemacht wird.

Das Beispielprogramm soll zwei neue Befehle in das BASIC 7.0 einführen:

merge

und

old

Die Wirkungsweise der beiden Befehle ist wahrscheinlich schon dem Namen nach bekannt. Der Merge-Befehl hängt ein BASIC-Programm an ein schon im Speicher befindliches an, der Old-Befehl hebt die Wirkung des New-Befehls wieder auf, sofern der Programmtext nicht durch anderweitige Einflüsse zerstört wurde.

Warum diese beiden häufig gebrauchten Befehle nicht schon Bestandteil des BASIC 7.0 sind, wird wahrscheinlich ein ewiges Geheimnis von Commodore oder Microsoft bleiben. Allerdings bietet uns das Fehlen der Befehle die Möglichkeit, ein praktisch brauchbares Beispiel zu entwerfen.

Ein dritter Befehl kommt dabei zu merge und old hinzu: »off« schaltet die Befehlserweiterung ab, die Vektoren werden auf ihren alten Stand gebracht.

```
         .base $1300

.define verck    = $0c     ; Load/Verify-Flag
.define txttab   = $2d     ; Programmstart
.define syntax1  = $80     ; Syntax-Checkbyte von DOSPAR
.define dossa    = $11d    ; Sekundäradresse im DOS-Puffer

.define iesclk   = $30c    ; Vektor: Token-Bildung
.define oldilk   = $4321   ; alte Vektoradresse

.define iescpr   = $30e    ; Vektor: Token listen
.define oldipr   = $51cd   ; alte IESCPR-Adresse

.define iescex   = $310    ; Vektor: Befehl ausführen
.define oldiex   = $4ba9   ; alter Vektor

.define txttop   = $1210   ; Zeiger auf Programmende

.define setvec   = $4251   ; alle Vektoren restaurieren
.define off      = setvec  ; = OFF-Befehl
.define search   = $43e2   ; Suche nach Befehlswort
.define setend   = $4f82   ; TXTTOP := INDEX+2
.define ustlst   = $516a   ; Befehlswort listen
.define loadin   = $913c   ; Einsprung in BASIC-LOAD
.define dospar   = $a3c3   ; Parameter für Load etc. holen
.define comcpl   = $a667   ; Befehls-String zusammensetzen
.define mistake  = $4d76   ; Fehlermeldung ausgeben
.define link     = $af87   ; Linkadressen erzeugen
```

```
.define cr       = $ff00   ; Konfigurationsregister
.define setbnk   = $ff68   ; Bank für Filename+LSV-Oper.

; Initialisierung der Befehlserweiterung
; Start durch bank15 : sys dec("1300")

init     lda #<(ownlk)   ; Vektoren auf eigene Routinen
         ldx #>(ownlk)   ; richten
         sta iesclk
         stx iesclk+1
         lda #<(ownpr)
         ldx #>(ownpr)
         sta iescpr
         stx iescpr+1
         lda #<(exec)
         ldx #>(exec)
         sta iescex
         stx iescex+1
         rts

; Tabelle der Befehlsworte

tab      .byte "mergE"
         .byte "olD"
         .byte "ofF",0

; Tabelle der Adressen der neuen Befehle

adr      .word merge-1
         .word old-1
         .word off-1

; Umwandlung der neuen Befehlsworte in Token

ownlk    tax             ; letztes Zeichen retten
         lda #>(tab)     ; Highbyte Tabellenadresse
         ldy #<(tab)     ; Lowbyte
         jsr search      ; Suche nach Befehlswort
         bpl nfound      ; nicht drin: Sprung
         and #$7f        ; drin: Bit 7 löschen
         clc             ; $27 addieren =
```

```
         adc #$27        ; zweites Tokenbyte
         ldx #0          ; Flag: Präfix $fe
         bcc back        ; immer!

nfound   sec             ; Flag: nicht gefunden
         txa             ; Akku wiederherstellen
back     jmp oldilk      ; zum Interpreter zurück

; Listen der neuen Befehle

ownpr    cpx #$ff        ; Flag für Präfix $ce?
         beq not         ; ja: Sprung
         cmp #$27        ; 2. Byte kleiner $27?
         bcc not         ; ja: Skip
         cmp #$2a        ; 2. Byte größer $2a?
         bcs not         ; ja: Ende
         sbc #$26        ; sonst: $27 subtr. (Carry=0!)
         ora #$80        ; 128 addieren
         tax             ; als Nummer in X übergeben
         lda #>(tab)     ; Highbyte Tabellenstart
         ldy #<(tab)     ; Lowbyte
         jsr ustlst      ; Befehlswort listen
not      sec             ; Flag: Interpreter nicht listen
         jmp oldipr      ; zurück zum Interpreter

; Befehlsadressen zu den neuen Token ermitteln und Befehls-
; routinen aufrufen

exec     cmp #2          ; 2. Tokenbyte kleiner 2?
         bcc nobef       ; ja: kein Befehl
         cmp #$2a        ; 2. Byte größer $2a?
         bcs nobef       ; ja: kein Befehl
         sbc #$26        ; sonst: $27 subtr.=Nummer 0-...
         asl             ; mal 2: 0,2,4,6,...
         tax             ; als Zeiger nach X
         lda adr+1,x     ; Highbyte der Befehlsadresse
         pha             ; auf den Stack
         lda adr,x       ; Lowbyte der Befehlsadresse
         pha             ; auf den Stack
         clc             ; Flag: User-Adresse auf Stack
         .byte $24       ; Bit-Opcode: 'sec' überlesen
```

```
nobef   sec              ; Flag: kein Usertoken
        jmp oldiex       ; zum Interpreter zurück

; Merge-Befehl: Programm anhängen

merge   lda #0           ; Load/Verify-Flag auf Load
        sta verck        ; setzen
        lda #%1110'0110 ; erlaubt Parameter wie bei DLOAD
        jsr dospar       ; Parameter holen
        lda syntax1      ; Checkbyte 1 von DOSPAR
        and #%0000'0001 ; 1. Dateinamen-Bit
        cmp #%0000'0001 ; gesetzt?
        bne synerr       ; nein: kein Name ist error
        lda #0
        sta cr           ; I/O einschalten!!!
        sta dossa        ; Sekundäradresse 0 für Load
        lda #4           ; Parameter für Zusammenbau
        ldy #5           ; des Load-Befehls-Strings
        jsr comcpl
        lda #0           ; Bank für Filename
        tax              ; Bank für Load/Save/Verify
        jsr setbnk       ; setzen
        lda txttop       ; Zeiger auf Programmende
        ldy txttop+1
        sec
        sbc #2           ; minus 2
        bcs m1
        dey
m1      tax              ; als Load-Start nach (X,Y)
        lda verck        ; Flag für Load
        jmp loadin       ; in Routine für LOAD einspringen

synerr  ldx #11          ; Nummer für syntax error
        jmp mistake      ; Meldung ausgeben

; Old-Befehl: Aufheben des Befehls New

old     ldy #1           ; Position Highbyte Linker
        tya              ; Akku ungleich 0
        sta (txttab),y   ; In Highbyte des Linkers der
                         ; ersten Programmzeile
        jsr link         ; Link-Adressen erzeugen
        jmp setend       ; Programmende = Link-Zeiger+2
```

Die Wirkungsweise der einzelnen Befehle, soweit dies nicht aus dem Kommentar selbst ersichtlich ist:

Der Befehl OFF ist identisch mit der Routine SETVEC des Interpreters, die sämtliche BASIC-Vektoren aus dem ROM in die erweiterte Zeropage kopiert. Damit wird die Befehlserweiterung abgeschaltet, da alle alten Vektorinhalte restauriert werden.
Der MERGE-Befehl macht sich wesentlich die Routine des BASIC-Befehls LOAD zunutze. Das Laden selbst sowie alle dabei nötigen Fehlerabfragen erfolgen in der LOAD-Routine. Die einzigen Änderungen bei MERGE gegenüber DLOAD bestehen in der Wahl der Startadresse des Ladens. Während DLOAD den Zeiger TXTTAB, den Start des BASIC-Programmspeichers, als Ladeadresse benutzt, wird bei MERGE einfach das Ende des Programmtextes dafür hergenommen. Da am Ende eines BASIC-Programms immer zwei zusätzliche Nullen stehen, müssen vom Zeiger TXTTOP noch zwei subtrahiert werden, um die richtige Ladeadresse zu erhalten.
Der OLD-Befehl benutzt die LINK-Routine, um im Speicher befindliche BASIC-Zeilen wieder aneinanderzubinden und gleichzeitig das Ende des Textes zu bestimmen. NEW bewirkt, daß in der ersten Programmzeile, auf die TXTTAB weist, das Highbyte des Linkers auf Null gesetzt wird. Das Programmende wird entsprechend herabgesetzt. Die LINK-Routine erzeugt die notwendigen Linkadressen soweit, bis ein Linker-Highbyte gleich Null gefunden wird. Dadurch, daß das von NEW gelöschte Link-Highbyte der ersten Zeile auf einen Wert ungleich Null gebracht wird, kann man hoffen, daß LINK erst durch das ursprüngliche Programmende aufgehalten wird, sofern das Programm nicht anderweitig zerstört wurde. Ist das Programmende erreicht, zeigt der Hilfszeiger INDEX der LINK-Routine genau hinter das Nullbyte, das das Ende der letzten Programmzeile anzeigt. Die SETEND-Routine addiert nun einfach zwei zum Zeiger INDEX und legt das Ergebnis in TXTTOP ab.

6.2 Die Vektoren ICRNCH, IQPLOP ($304, $306)

Die beiden Vektoren

ICRNCH = **$304/ $305** ($430d)

und

IQPLOP = **$306/ $307** ($5151)

liegen wie die beiden Vektoren IESCLK und IESCPR innerhalb der TOKEN- beziehungsweise der LIST-Routine (Normalbelegung in Klammern).

Während mit IESCLK und IESCPR allerdings nur die Erzeugung beziehungsweise das Listen von eigenen Token beziehungsweise Befehlsworten möglich ist, bieten die beiden vorliegenden Vektoren weitergehende Möglichkeiten, da der Sprung über die Vektoren gleich der erste Befehl der entsprechenden Routinen ist.
Allerdings muß ich gestehen, daß mir keine praktischen Beispiele für den Einsatz der beiden Vektoren einfallen, jetzt, da im Gegensatz zum Commodore 64 die Einführung von Usertoken möglich ist. Es sei denn, man setzt die Vektoren einfach auf ein rts, so daß einmal die Bildung von Token, zum anderen das Listen überhaupt verhindert wird.

6.3 Der Vektor IERROR ($300, normal: $4d3f)

Bei diesem Vektor kann man sich den praktischen Einsatz sehr leicht vorstellen. Er liegt nämlich zu Beginn der Routine, die für die Ausgabe von Fehlermeldungen zuständig ist. Wird eine zusätzliche Fehlermeldung gebraucht, zum Beispiel »can't merge«, wenn noch kein Programm im Speicher steht und das Mergen ohne nicht funktionieren soll, kann man den IERROR-Vektor auf ein eigenes Programm umbiegen, das alle alten Fehlernummern an die alte Routine und neue Fehlermeldungen selbst ausgibt.
Ich glaube, hierfür ist kein eigenes Beispiel notwendig.

6.4 Der Vektor IMAIN ($302, $4dc6)

Dies ist wieder einer der interessanteren Vektoren: er liegt gleich zu Beginn der Schleife, die im Direktmodus Eingabezeilen in den Eingabepuffer holt. Innerhalb derselben Schleife wird auch das Einfügen und Löschen von Programmzeilen vorgenommen, auf diese Tatsache stützt sich das folgende Beispiel.
Es soll das Strukturieren eines BASIC-Programmtextes dadurch unterstützen, daß Leerzeichen am Zeilenstart nicht mehr eliminiert werden. Eine Strukturierung läßt sich zwar auch durch die Eingabe von Doppelpunkten oder Großbuchstaben am Zeilenbeginn erreichen, das Beispiel ist aber trotzdem auch als Utility ganz gut zu gebrauchen.

```
        .base $1300

    .define curlin  = $3b     ; aktuelle Zeilennummer
    .define txtptr  = $3d     ; CHRGET-Programmzeiger
    .define skip    = $395    ; zu überl. Zeichen bei CHRGET
```

```
.define imain   = $302    ; Vektor: Direktmodus-Schleife

.define mainin1 = $4dd9   ; Einsprung in die alte Routine für
                          ; den Fall, daß keine Zeilennummer
                          ; am Zeilenstart steht
.define mainin2 = $4de5   ; Einsprung für Zeile einfügen/
                          ; löschen

.define getlin  = $4f93   ; Eingabezeile holen
.define linnum  = $50a0   ; Zeilennummer holen

init      lda #<(own)     ; IMAIN-Vektor auf eigene
          ldx #>(own)     ; Routine richten
          sta imain
          stx imain+1
          rts

own       ldx #$ff        ; Flag: Direktmodus
          stx curlin+1
          jsr getlin      ; Eingabezeile holen
          stx txtptr      ; CHRGET-Zeiger auf Pufferstart
          sty txtptr+1
          jsr chrget      ; 1. Zeichen ungleich Blank
          beq own         ; Zeile leer: Schleife
          bcc own1        ; Ziffer: weiter
          jmp mainin1     ; sonst: in die alte Routine

own1      ldx #$ff        ; Blanks nicht überlesen
          stx skip        ; Akku = Ziffer muß bleiben
          jsr linnum      ; Zeilennummer holen
          lda #" "        ; Blanks wieder überlesen
          sta skip
          jmp mainin2     ; Zeile einfügen, löschen
```

In diesem Beispiel wird die Trickkiste geöffnet, um zu erreichen, daß die Blanks am Zeilenstart nicht mehr vernichtet werden: Das normale Überlesen der Leerzeichen wird durch einen direkten Eingriff in die CHRGET-Routine verhindert. Siehe hierzu auch Kapitel 2.2.1! Das $ff, das in die CHRGET-Adresse SKIP gespeichert wird, entspricht dem BASIC-Token für Pi, das bestimmt nicht am Anfang einer Programmzeile steht.

Die beiden Einsprungadressen in die alte MAIN-Routine erklären sich aus einer genaueren Kenntnis der betreffenden Routine. Solche Kunstgriffe sind nur dann möglich, wenn man Routinen entweder unter der Verwendung eines Disassemblers oder mit einem ROM-Listing vorher analysiert hat.

6.5 Der Vektor IGONE ($308, normal: $4aa2)

IGONE liegt wie IESCEX in der Hauptschleife des Interpreters. Er verhält sich ähnlich zu IESCEX wie ICRNCH zu IESCLK oder IQPLOP zu IESCPR. Der Sprung über IGONE ist die erste Anweisung innerhalb der Hauptschleife.
Die Entsprechung dieses Vektors wurde beim 64er noch zur Einbindung neuer BASIC-Befehle in den Interpreter benutzt. Jetzt, da uns das BASIC-7.0 die Usertoken anbietet, wird der Vektor voraussichtlich keine so große Rolle mehr spielen.
Tatsächlich fällt mir auch kein Beispiel ein, das sich nicht in anderer Weise eleganter lösen ließe als mit dem Vektor IGONE. Nun, vielleicht findet sich aber doch einmal die Gelegenheit, mit Hilfe des Vektors eine Spezialfunktion zu erfüllen, man wird sehen.

6.6 Der Vektor IEVAL ($30a, normal: $780a)

Dies ist wieder ein Vektor mit zugehörigem Beispiel. Der IEVAL-Vektor liegt am Start der EVAL-Routine, jener Routine, die für FRMEVL die einzelnen Terme eines Ausdrucks holt (Kapitel 5.6).
Während über den IESCFN-Vektor lediglich die Einführung neuer Funktionen zu machen war, bietet uns der neue Vektor, der übrigens schon beim 64er zu finden war, die Möglichkeit eines noch grundsätzlicheren Eingriffs.
Als Beispiel hierfür habe ich die Einführung sedezimaler Zahlen in das BASIC vorgesehen. Diese sollen wie beim Monitor durch ein vorangestelltes Dollarzeichen von dezimalen Zahlen unterschieden werden. Beispielsweise $c000 für 49152 usw.
Das Beispiel schien mir ganz nützlich, da die Benutzung der DEC-Funktion zur Umrechnung der sedezimalen in dezimale Zahlen doch auf die Dauer sehr umständlich ist.

```
     .base $1300

   .define valtyp  = $0f     ; Werttyp-Flag
   .define fac1    = $63     ; FAC
```

```
.define ieval   = $30a   ; Vektor: Term holen
.define chrget  = $380   ; CHRGET
.define evalin  = $78e1  ; Einsprung in die EVAL-Routine
.define normint = $af0f  ; Umwandlung Integer nach Fließk.
.define mistake = $4d76  ; Fehlermeldung angeben

init    lda #<(own)      ; Vektor auf eigene Routine
        ldx #>(own)
        sta ieval
        stx ieval+1
        rts

own     lda #0           ; Flag Werttyp auf
        sta valtyp       ; numerisch
        jsr chrget       ; 1. Zeichen des Terms holen
        php              ; Carry-Flag retten
        cmp #"$"         ; war Zeichen '$' ?
        beq hexcons      ; ja: Hexzahl holen
        plp              ; Status wiederherstellen
        jmp evalin       ; zurück zur EVAL-Routine

hexcons lda #0           ; Übergabezellen löschen
        sta count        ; Zähler für Ziffern
        sta fac1+1
        sta fac1+2
        pla              ; Status vom Stack holen
loop    jsr chrget       ; sedezimale Ziffer
        bcc m1           ; Dezimalziffer: Sprung
        cmp #"a"         ; kleiner ASCII 'a' ?
        bcc end          ; ja: nicht gültig
        cmp #"f"+1       ; größer ASCII 'f' ?
        bcs end          ; ja: nicht gültig
        sbc #7           ; Binärwert der Ziffer herstellen
m1      sbc #"0"-1       ; ASCII '0' minus 1 subtrahieren
        ldx #3           ; Zähler für Multiplikation
m2      asl fac1+2       ; Zahl * 2
        rol fac1+1       ; 4 mal gemacht = * 16
        dex              ; Zähler - 1
        bpl m2           ; nicht negativ: Schleife
        ora fac1+2       ; Sedezimale Ziffer einodern
        sta fac1+2       ; und wieder abspeichern
```

```
        inc count
        lda count
        cmp #5          ; Ziffernzähler: schon 4 Ziffern?
        bcc loop        ; nein: nächste Ziffer
        ldx #15         ; ja: Nummer für overflow
        jmp mistake     ; Meldung ausgeben

end     sec             ; Flag: kein 2er-Kompl. bilden
        ldx #$90        ; Exponent  16
        jmp normint     ; Umwandeln in Fließkomma

count   .byte 0
```

7
Die Belegung der Zeropage

Adresse		Name	Erläuterung
$00	0	d8502	Datenrichtungsregister zum folgenden Prozessor-Port. Jedes gesetzte Bit schaltet das entsprechende Bit von r8502 als Ausgang.
$01	1	r8502	Prozessor-Port. Der Prozessor-Port ist keine normale Zeropage-Speicherzelle, sondern ein Register, wie es zum Beispiel auch von den CIAs schon bekannt ist. Der Port stellt eine direkte Verbindung zum Prozessor dar. Die einzelnen Bits haben die folgende Bedeutung: Bit 7 nicht benutzt 6 1 = ASCII/DIN-Taste gedrückt 5 0 = Kassettenmotor ein 4 0 = PLAY gedrückt 3 1 = Datenleitung Kassette = 1 2, 1: 01 = Textmodus; 10 = Grafikmodus 0: 1 = Farb-RAM Text; 0 = Farb-RAM Multicolorfarbe 2 (D800-DC00)
$02	2	bank	Konfigurationsindex für Sprünge mit JSRFAR. Konfiguration bei der Registeranzeige des Monitors (es wird nur das untere Halbbyte angezeigt, wenn die Registeranzeige durch ein Break erreicht wurde!)

$03	3	pchi	Oberes Halbbyte der Sprungadresse bei JMPFAR und JSRFAR. Oberes Halbbyte der Break-Adresse+2 bei einer Registeranzeige durch BRK)
$04	4	pclo	Dasselbe für das untere Halbbyte
$05	5	sreg	Zwischenspeicher für das Prozessor-Status-Register für JSRFAR, JMPFAR und Registeranzeige
$06	6	areg	Akkumulator für denselben Zweck
$07	7	xreg	X-Register
$08	8	yreg	Y-Register
$09	9	stkptr	Stackpointer

Bereich für BASIC:

$09	9	charac	Zwischenspeicher für gesuchte Zeichen
$0a	10	endchr	Zwischenspeicher für ein am Ende einer Zeichenkette gesuchtes Zeichen, zum Beispiel ein Anführungszeichen
$0b	11	trmpos	Zwischenspeicher für den TAB-Befehl
$0c	12	verck	BASIC-Flag für LOAD und VERIFY (LOAD = 0)
$0d	13	count	Zeiger in den Eingabepuffer. Länge einer eingegebenen Zeile. Zwischenspeicher für dimensionierte Variable
$0e	14	dimflg	Flag für Dimensionierung durch DIM-Befehl
$0f	15	valtyp	FRMEVL-Werttypus. $ff = String / $00 = numerisch
$10	16	intflg	Flag für Integer- ($80) beziehungsweise Realzahlen ($00)
$11	17	dores	Allgemeiner Zwischenspeicher zum Beispiel für Garbage Collection oder LIST
$12	18	subflg	Flag für FN-Aufrufe
$13	19	inpflg	Flag für INPUT ($00), GET ($40) oder READ ($98)
$14	20	domask	Maske für Vergleichsoperationen. Flagge für Vorzeichen trigonometrischer Funktionen
$15	21	channl	Aktuelles BASIC-I/O-Gerät
$16/$17	22	kinnum	Zeilennummer oder anderer Integerwert (LIST, GOTO, GOSUB, PEEK, POKE ...)
$18	24	temppt	Zeiger auf nächsten freien Platz im temporären Stringstack ab $1b
$19/$1a	25	lastpt	Zeiger auf letzten Eintrag im temporären Stringstack. Da dieser in der Zeropage liegt, ist $1a immer Null

$1b-$23	27	tempst	Temporärer Stringstack. In diesen insgesamt 9 Byte in der Zeropage wird praktisch die Verwaltung aller auftretenden Strings vorgenommen. Die Adressen der Strings landen alle im Stringstack
$24-$27	36	index	Bereich für allgemeine Hilfszeiger
$28-$2c	40	res	Arbeitsspeicher für Arithmetik. Zwischenspeicher für Multiplikation und Division
$2d/$2e	45	txttab	Zeiger auf den Start des BASIC-Programmtextes. Normalwert ist $1c00, wenn keine Grafik eingeschaltet ist
$2f/$30	47	vartab	Zeiger auf den Start der Variablen in Bank 1. Normalwert ist $400, das Ende der Common Area
$31/$32	49	arytab	Zeiger auf den Start der dimensionierten Variablen (gleichzeitig das Ende der nichtdimensionierten)
$33/$34	51	strend	Zeiger auf das Ende des String-Speichers in Bank 1. Gleichzeitig das Ende der dimensionierten Variablen, da die Strings in Bank 1 von normalerweise $ff00 nach unten wachsen
$35/$36	53	fretop	Zeiger auf den Start des String-Speichers (+1). Weist normalerweise auf $ff00 in Bank 1. Die Strings wachsen von dieser Adresse nach unten
$37/$38	55	frespc	Hilfszeiger für String-Übertragungen
$39/$3a	57	maxmem1	Höchste verfügbare Adresse für BASIC in Bank 1
$3b/$3c	59	curlin	Nummer der aktuellen Programmzeile im RUN-Modus. Ist $3c gleich $ff, wird dies als Zeichen für den Direktmodus gewertet
$3d/$3e	61	txtptr	Zeiger in den BASIC-Text für CHRGET
$3f/$40	63	fndpnt	Zeiger auf gefundenes Key-Wort für SEARCH. Hilfszeiger für PRINT USING
$41/$42	65	datlin	Nummer der aktuellen DATA-Zeile
$43/$44	67	datptr	Adresse der laufenden DATA-Zeile
$45/$46	69	inpptr	Adresse, von der INPUT, GET und READ Eingaben holt. Für READ identisch mit der vorigen Adresse
$47/$48	71	varnam	Name der aktuellen Variablen
$49/$4a	73	varpnt	Adresse des Wertes der Variablen in Bank 1
$4b/$4c	75	lstpnt	Zwischenzeiger für LIST, für die Laufvariable bei FOR/ NEXT, für AND, EOR, für Zuweisungen an Variable etc.
$4d/$4e	77	opptr	Zeiger auf nächste Operation bei FRMEVL. Bei READ Zwischenspeicher
$4f	79	opmask	Vergleichsmaske (kleiner=$04, gleich=$02, größer=$01)
$50/$51	80	defpnt	Zeiger auf FN-Variable

$52/$53	82	dscpnt	Zwischenspeicher für String-Operationen
$59-$5d	84	tempf1	Temporäres Fließkommaregister für Polynomberechnungen bei der Auswertung transzendenter Funktionen
$5e-$62	94	tempf2	Temporäres Register Nummer 2. Auch für die Polynomberechnung.
$63-$67	99	facl	Fließkommaregister 1. Exponent = $63, Mantisse = $64 bis $67, wobei $64 das höchstwertige Mantissen-Byte ist
$68	104	facsgn	Vorzeichen der Mantisse von FAC1
$69	105	sgnflg	Flag für Vorzeichen bei String-/Fließkommawandlung ($ff = Minus). Polynomgrad bei der Berechnung transzendenter Funktionen, daher auch der Name »degree«, der im Handbuch erwähnt ist.
$6a-$6e	106	arg	Fließkommaregister 2. Exponent = $6a, Mantisse von $6b bis $6e
$6f	111	argsgn	Vorzeichen der Mantisse von ARG
$70	112	arisgn	Vorzeichenvergleichs-Flag. $ff = FAC und ARG haben ungleiche Vorzeichen 0 = FAC und ARG haben gleiches Vorzeichen
$71	113	facov	Rundungsbyte zu FAC. Rundungsstellen treten bei fast jeder Operation auf.
$72/$73	114	fbufpt	Zeiger auf Kassettenpuffer. Dieser liegt normalerweise von $b00 bis $bff
$74/$75	116	autinc	Inkrement für den Befehl AUTO.
$76	118	mvdflg	Flag für Programmstart verschoben. Wird der VIC-Grafikschirm benutzt, wird der Programmstart um 9 KByte nach oben verschoben und dieses Flag gesetzt.
$77	119	sprnum	Sprite-Nummer bei MOVSPR-Befehl. Zwischenspeicher für Lautstärke bei VOL. Flag für Abbruch bei PRINT USING. Einfügestelle des neuen Strings bei der Zuweisung MID$=. Ein allgemeiner Zwischenspeicher für alle Zwecke also.
$78	120	hulp	Länge des einzubauenden Strings bei der Zuweisung MID$ =. String-Länge bei der Ausgabe der Funktionstasten-Strings. Zeiger in abzuspielenden String bei PLAY usw. usf. Die Speicherzelle hulp wird meist in Verbindung mit sprnum gebraucht. Wo hulp benutzt wird, ist sprnum auch eingesetzt.
$79	121	syntmp	Zwischenspeicher bei String-Vergleichen, beim Anlegen von String-Arrays und bei vielen anderen Operationen. Im allgemeinen dient diese Speicherstelle dann als

			Zwischenspeicher, wenn Vergleiche mit Werten aus anderen Bänken vorgenommen werden. Das Zeichen aus der anderen Konfiguration wird in syntmp zwischengespeichert.
$7a	122	dsdlen	Monitor: Zeiger in den Eingabepuffer ab $200. BASIC: Länge von DS$. Die Floppy-Statusvariable gilt als gelöscht, wenn DSDLEN gleich Null ist.
$7b/$7c	123	dsdadr	Adresse von DS$ im String-Bereich innerhalb der RAM-Bank 1. DS$ wird bei Aktionen mit der Floppy laufend aktualisiert.
$7d/$7e	125	tos	Zeiger auf das obere Ende des BASIC-Runtime-Stacks. Einträge in diesen User-Stack, der nichts mit dem Prozessor-Stack zu tun hat, erfolgen von oben nach unten wie im String-Bereich. tos ist der laufende Zeiger. Im Runtime-Stack werden Informationen über FOR/NEXT-Schleifen, DO/LOOP-Schleifen und RETURN-Adressen untergebracht. Beim C 64 geschah dies noch im Prozessor-Stack, dieser Bereich wäre aber für den 128er mit seinen erweiterten Fähigkeiten zu klein gewesen. Der Runtime-Stack reicht von $a00 herab bis $800.
$7f	127	runmod	Flag für Run- oder Direktmodus: $80 = Run-Modus, 0 = Direktmodus
$80	128	syntax1	Syntax-Checkbyte der DOSPAR-Routine: Bit 7 Klammeraffe gelesen (File überschreiben) 6 Recordlänge gelesen 5 2. Laufwerksnummer gelesen 4 1. Laufwerksnummer gelesen 3 Gerätenummer gelesen 2 logische File-Nummer gelesen 1 2. File-Name gelesen 0 1. File-Name gelesen
$81	129	syntax2	Syntax-Checkbyte 2: Bit 2 Endadresse gelesen 1 Startadresse gelesen 0 Segmentnummer gelesen Andere Bits haben keine Bedeutung.
$82	130	oldstk	Speicher für den Prozessor-Stackpointer beim TRAP-Befehl.

Zeropage für Grafikbefehle:

\$83	131	colsel	Farbquelle bei PAINT, DRAW, GSHAPE usw. Die Farbquelle kann Werte zwischen 0 und 3 annehmen. Die einzelnen Werte bezeichnen die Herkunft der aktuellen Farbe: 0 Farbe stammt vom Hintergrund 1 Farbe = Zeichenfarbe 2 Farbe = Multicolor 1 3 Farbe = Multicolor 2
\$84	132	mcolor1	Multicolorfarbe 1. Beim Einschalten oder RESET wird diese Farbe als weiß eingestellt (mcolor1 = 1).
\$85	133	mcolor2	Multicolor-Zusatzfarbe 2. Einschaltfarbe ist rot (mcolor2 = 2).
\$86	134	foregnd	Grafik-Pixelfarbe. Einschaltzustand: foregnd = 13 (hellgrün).
\$87/\$88	135	scalex	Skalierung in X-Richtung, durch SCALE-Befehl zu setzen.
\$89/\$8a	137	scaley	Skalierung in Y-Richtung.
\$8b	139	stopnb	Temporäres Flag für Flächenbegrenzungen bei PAINT.
\$8c/\$8d	140	grapnt	Allgemeiner Zeiger in den HiRes-Speicher.
\$8e	142	vtemp1	Allgemeines Byte für Zwischenspeicherungen.
\$8f	143	vtemp2	Allgemeines Byte für Zwischenspeicherungen.

Kernal-Zeropage:

\$90	144	status	IEC- und Kassetten-Status-Byte. Die einzelnen Bits haben die Bedeutung: Bit 7 gesetzt: device not present oder Bandende 6 = 1: End of File 5 = 1: Prüfsummenfehler 4 = 1: Fatal Error 3 = 1: Block zu lang 2 = 1: Block zu kurz 1 = 1: Timeout beim Lesen 0 = 1: Timeout beim Schreiben
\$91	145	stkey	Ergebnis der Tastaturabfrage auf die gedrückte STOP-Taste. Durch die spezielle Matrixabfrage werden auch

			andere Tasten als die STOP-Taste erfaßt. Der Wert von stkey beträgt beispielsweise: gedrückte STOP-Taste = $7f gedrückte CBM-Taste = $df gedrücktes Control = $fb
$92	146	svxt	Timing-Konstante für Kassettenbetrieb.
$93	147	verck	Load/Verify-Flag des Betriebssystems. 0 Load 1 Verify
$94	148	c3po	Flag für Zeichen im IEC-Puffer ($80 = Zeichen im Puffer).
$95	149	bsour	Puffer für seriellen Bus.
$96	150	syno	Synchronisationswert für Kassettenbetrieb.
$97	151	xsav	Wie der Name schon vermuten läßt, ein Zwischenspeicher für die Register X und Y.
$98	152	ldtnd	Anzahl der Einträge in die Tabellen der logischen Files = Anzahl offener Files.
$99	153	dfltn	Aktuelles Eingabegerät (Default-Wert 0 = Tastatur).
$9a	154	dflto	Aktuelles Ausgabegerät (Default-Wert 3 = Bildschirm).
$9b	155	prty	Paritäts-Byte bei Kassettenbetrieb.
$9c	156	dpsw	Flag für Byte-Ein-/Ausgabe bei Kassettenbetrieb.
$9d	157	msgflg	Flag für die Ausgabe von Systemmeldungen. Bit 7 ist zuständig für die Ausgabe von Fehlermeldungen der Art »I/O error #«, Bit 6 erlaubt oder verbietet die Ausgabe der Meldungen der Art »loading« oder »searching for«. Gesetzte Bits erlauben die Ausgabe der entsprechenden Meldungen des Betriebssystems.
$9e	158	ptr1	Auszugebendes Zeichen bei Kassettenbetrieb. Zwischenspeicher für Sekundäradresse bei LOAD vom IEC-Bus. 1-Byte-Zeiger in den Kassettenpuffer beim Booten von Diskette. Falsch gelesenes Byte von Kassette etc.
$9f	159	ptr2	1-Byte-Zeiger in den Kassettenpuffer beim Schreiben auf Kassette. Korrigiertes Byte beim Lesen von Kassette. Zwischenspeicher für Länge des File-Namens bei Load vom IEC-Bus usw.
$a0-$a2	160	time	3-Byte-Systemuhr. Die Uhr wird während des Interrupts hochgezählt, der ca. jede $1/60$ Sekunde erfolgt. Sie ist keine Echtzeituhr wie im Handbuch behauptet. $a0 ist das höchstwertige Byte der Uhr.

$a3	163	r2d2	Temporäres Flag bei der Ausgabe auf den IEC-Bus und bei Kassettenbetrieb.
$a4	164	bsourl	Temporäres Datenregister für Eingabe vom IEC-Bus. Flag für Datenempfang beim Kassettenbetrieb. Bitzähler für Bandausgabe.
$a5	165	count	Bitzähler für IEC-Ausgabe und Bandausgabe.
$a6	166	bufpt	Einbyte-Zeiger in den Kassettenpuffer.
$a7	167	inbit	Empfangene Bits bei RS232-Übertragung. Blockzahl beim Lesen beziehungsweise Schreiben von beziehungsweise auf Kassette.
$a8	168	bitci	Bitzähler für empfangene Bits bei RS232-Übertragung. Zeiger für Lesefehler bei Bandbetrieb.
$a9	169	rinone	Flag für Startbit bei RS232-Übertragung. Auch beim Bandbetrieb verwendet, um einen Wechsel der Impulslängen anzuzeigen.
$aa	170	ridata	RS232- und Band-Datenpuffer.
$ab	171	riprty	Paritätsbyte RS232 und Bandbetrieb.
$ac/$ad	172	sal	Zeiger in den Kassettenpuffer. Speicher für Startadresse eines Maschinenprogramms beim schnellen Laden vom IEC-Bus = Startadresse für BASIC-Befehl BOOT. Leider beim langsamen Laden vergessen, dadurch funktioniert der BOOT-Befehl nur mit einer 1570 oder 1571.
$ae/$af	174	eal	Laufender Zeiger bei LOAD. Enthält nach dem Laden die Endadresse. Bei SAVE: Endadresse des abzuspeichernden Bereichs.
$b0	176	cmpo	Zähler für Zeit bei Bandbetrieb.
$b1	177	cmpok	Zeitkonstante für Bandbetrieb.
$b2/$b3	178	tapel	Zeiger auf Start des Kassettenpuffers. Normalwert ist $b00.
$b4	180	bitts	Bitzähler für RS232-Übertragung. Flag Timer A frei für Bandbetrieb.
$b5	181	nxtbit	Nächstes auszugebendes Bit bei RS232-Übertragung. Flag für gültiges EOB bei Bandbetrieb.
$b6	182	rodata	RS232: auszugebendes Byte. Bandbetrieb: Flag für Byte-Lesefehler.
$b7	183	fnlen	Länge des aktuellen File-Namens.
$b8	184	la	Aktuelle logische File-Nummer.
$b9	185	sa	Aktuelle Sekundäradresse.
$ba	186	fa	Aktuelle Geräteadresse.
$bb/$bc	187	fnadr	Adresse des aktuellen File-Namens.

$bd	189	roprty	RS232-Ausgabebyte-Parität. Zwischenspeicher für VERIFY.
$be	190	fsblck	Anzahl Blöcke, die bei Bandbetrieb zu lesen oder schreiben sind.
$bf	191	drive	Zwischenspeicher für Lesen vom Band. ASCII-Wert »0« für Laufwerk beim Booten.
$c0	192	casl	Flag: Recordermotor eingeschaltet. Enthält den Wert des Prozessor-Ports.
$cl/$c2	193	sta	Startadresse für Datenblock auf Band schreiben. Ladezeiger für Load von Band. Startadresse des abzuspeichernden Bereichs bei Save. $cl = Sektor für Boot, Sc2 = Track für Boot.
$c3/$c4	195	memuss	Zwischenspeicher für Kopieren der Kernal-Vektoren in die erweiterte Zeropage. ZSP für Suche nach »CBM« in Bank 1. Startadresse für Load und Verify.
$c5	197	data	Bitspeicher für Lesen von Band. Prüfsummenpuffer für Schreiben auf Band.
$c6	198	ba	Konfigurationsindex für Load, Save und Verify.
$c7	199	fnbank	Konfigurationsindex, unter dem der aktuelle File-Name zu finden ist.
$c8/$c9	200	ribuf	Startadresse des RS232-Eingabepuffers. Normalwert ist $c00, der Eingabepuffer reicht damit von $c00 bis $d00 (in Bank 0).
$ca/$cb	202	robuf	Startadresse des RS232-Ausgabepuffers. Normalwert ist $d00, der Puffer geht also von $d00 bis $e00 (in der RAM-Bank 0).

Bereich für die Bildschirmsteuerung und Tastaturroutinen:

$cc/$cd	204	keytab	Zeiger auf aktuelle Tastatur-Decodiertabelle. Enthält einen Wert aus den Vektoren auf die möglichen Tastatur-Decodiertabellen ($33e/$33f und folgende).
$ce/$cf	206	imparm	Reservierter Zeiger für die Kernal-Routine PRIMM.
$d0	208	ndx	Anzahl Zeichen im Tastaturpuffer.
$dl	209	kyndx	Länge eines Funktionstasten-Strings. Gleichzeitig Flag für das Holen von Zeichen aus dem Funktionstasten-String, wenn ungleich Null.
$d2	210	keyidx	1-Byte-Zeiger auf aktuellen Funktionstasten-String. Weist in den Bereich ab $100a.

$d3	211	shfflg	Aktuelles Shift-Muster: shfflg — gedrückte Taste % 0000 0000 — keine % xxxx xxx1 — Shift % xxxx xx1x — CBM (C=) % xxxx x1xx — Control % xxxx 1xxx — ALT % xxx1 xxxx — DIN-Tastaturbelegung
$d4	212	sfdx	Tastennummer der aktuell gedrückten Taste (Zuordnung der Tastennummern s. Anhang).
$d5	213	lstx	Speicher für letzte Tastennummer für Repeat-Logik.
$d6	214	crsw	Flag: Lesen vom Bildschirm mit oder ohne eingeschalteten Cursor. crsw ungleich Null entspricht einem Lesen bei eingeschaltetem Cursor.
$d7	215	mode	Flag: 40- oder 80-Zeichen-Bildschirm. 40 Zeichen = 0, 80 Zeichen = $80.
$d8	216	graphm	Flag: Grafikmodus. graphm — Bildschirm % 0000 0000 — Textschirm % 0010 0000 — HiRes-Schirm % 0110 0000 — Splitscreen HiRes/Text % 1010 0000 — gesamter Schirm Multicolor % 1110 0000 — Splitscreen Multicolor/Text
$d9	217	charen	Flag: Zeichengenerator im ROM oder im RAM. Dieses Flag wird nur für den VIC benötigt, da der VDC die Zeichen sowieso in sein eigenes RAM übernimmt. (0 = Zeichen im ROM, %0000 0010 = Zeichengenerator im RAM).
$da/$db	218	sedsal	Hilfszeiger des Bildschirmeditors. Beispielsweise für das Scrollen oder Löschen von Zeilen oder die Definition von Funktionstasten.
$dc/$dd	220	sedeal	Zweiter Hilfszeiger.
$de	221	sedt1	Temporärer Byte-Speicher des Kernal-Editors.
$df	222	sedt2	Zweiter Byte-Speicher.

Aktiver Bildschirmspeicher:

$e0/$e1	224	pnt	Startadresse der aktuellen Bildschirmzeile innerhalb des jeweiligen Video-RAMs.

$e2/$e3	226	user	Startadresse der aktuellen Bildschirmzeile innerhalb des Farb- oder Attribut-RAMs.
$e4	228	scbot	Untere Grenze des Bildschirmfensters (gesamter Bildschirm = 24).
$e5	229	sctop	Obere Grenze des Bildschirmfensters (Gesamtschirm = 0).
$e6	230	sclf	Linke Begrenzung des Bildschirmfensters (Gesamtschirm = 0).
$e7	231	scrt	Rechte Fenstergrenze (Gesamtschirm = 39 beziehungsweise 79).
$e8	232	lsxp	Startspalte für Eingabe bei blinkendem Cursor.
$e9	233	lstp	Startzeile für Eingabe bei blinkendem Cursor.
$ea	234	indx	Letzte Spalte für das Lesen von Zeichen vom Bildschirm.
$eb	235	tblx	Aktuelle Cursor-Position: Zeilennummer.
$ec	236	pntr	Aktuelle Cursor-Spalte.
$ed	237	lines	Höchste Zeilennummer (24).
$ee	238	columns	Höchste Spaltennummer (39 oder 79).
$ef	239	datax	Nächstes auszugebendes Zeichen.
$f0	240	lstchr	Letztes ausgegebenes Zeichen. Wird gespeichert, um die Behandlung von Escape-Sequenzen zu ermöglichen.
$f1	241	color	Aktuelle Schriftfarbe beziehungsweise Attribut.
$f2	242	tcolor	Zwischenspeicher.
$f3	243	rvs	Reverse-Flag: Ist das Flag ungleich Null, ist der Revers-Modus eingeschaltet.
$f4	244	qtsw	Quotation-Flag: Flag für die Ausgabe von Steuerzeichen in Form reverser Zeichen. Ist der Wert des Flags ungleich Null, ist der Quote-Modus eingeschaltet.
$f5	245	insrt	Zähler für eingegebene Inserts.
$f6	246	insflg	Flag: Automatischer Einfügemodus (ESC + »a«). $ff = Modus ein.
$f7	247	locks	Flag: Verhinderung der Zeichensatzumschaltung mit Shift + Commodore, Verhindern der Bildschirmausgabe-Pause mit NO SCROLL oder Control + »s«. Bit Wirkung 7 = 1: sperrt Shift/Commodore 6 = 1: sperrt NO Scroll Andere Bits haben keine Bedeutung.
$f8	248	scroll	Flag: Sperren des Bildschirm-Scrollens durch Cursor-Bewegungen.

			Bit Bedeutung
			7 = 1: Verhindert Scrollen
			6 = 1: Verhinderung der Zeilenverknüpfung
			Siehe auch Kapitel 3.2.4.
$f9	249	beeper	Flag: Sperren von Control + »g« (Signalton) durch Setzen des Bits 7 möglich.
$fa-$ff	250		Freie Zeropage-Adressen zur Nutzung in eigenen Maschinenprogrammen!

7.1 Die erweiterte Zeropage

Adresse		Name	Erläuterung
$100	256	fbuffr	Speicherbereich für Umwandlung von Fließkommazahlen in ASCII-Strings.
$110	272	xcnt	Zwischenspeicher.
$111	273	dosf1l	Länge des ersten File-Namens.
$112	274	dosds1	Erste Laufwerknummer.
$113	275	dosf2l	Länge des zweiten File-Namens.
$114	276	dosds2	Zweite Laufwerknummer.
$115/$116	277	dosf2a	Adresse des zweiten File-Namens.
$117/$118	279	dosofl	Startadresse.
$119/$11a	281	dosofh	Endadresse.
$11b	283	dosla	Logische File-Nummer.
$1c	284	dosfa	Gerätenummer.
$11d	285	dossa	Sekundäradresse.
$11e	286	dosrcl	Recordlänge.
$11f	287	dosbnk	Konfigurationsindex für LSV-Operationen.
$120/$121	288	dosdid	Zwei ASCII-Zeichen für ID.
$122/$123	290	didchk	Flag: ID eingegeben.

Zu diesen Speicherzellen siehe auch bei den Routinen DOSPAR und COMCPL in Kapitel 5.7.

Der untere Bereich des Prozessor-Stacks wird auch vom BASIC-Befehl PRINT USING benutzt. Da die Routinen dieses Befehls aber kaum für eigene Zwecke zu gebrauchen sind, möchte ich die von PRINT USING benutzten Bytes in diesem Bereich nicht aufführen.

$137-$1ff	311	sysstk	Unteres Ende des Prozessor-Stacks. Tiefer darf der Stackpointer nicht gehen, sonst werden Speicherstellen von DOSPAR oder PRINT USING überschrieben.
$200-$2a1	512	buf	Eingabepuffer für BASIC und Monitor. Länge dieses Puffers ist 160 Byte = zwei Bildschirmzeilen des 80-Zeichen-Schirms.

Kernal-Routinen der Common Area:

$2a2-$2ae	674	fetch	FETCH-Routine
$2af-$2bd	687	stash	STASH-Routine
$2be-$2cc	702	cmpare	CMPARE-Routine
$2cd-$2e2	717	jsrfar	JSRFAR-Routine
$2e3-$2fb	739	jmpfar	JMPFAR-Routine

BASIC-Vektoren:

$2fc/$2fd	764	iescfn	Vektor: User-Funktion auswerten ($4c78).
$2fe/$2ff	766	bnkvec	Nicht benutzt.
$300/$301	768	ierror	Vektor: Fehlermeldung ausgeben ($4d3f).
$302/$303	770	imain	Vektor: Direktmodus ($4dc6).
$304/$305	772	icrnch	Vektor: Token-Bildung ($430d).
$306/$307	774	iqplop	Vektor: Befehlswort listen ($5151).
$308/$309	776	igone	Vektor: Interpreter-Hauptschleife ($4aa2).
$30a/$30b	780	ieval	Vektor: Term holen ($78da).
$30c/$30d	782	iesclk	Vektor: Usertoken bilden ($4321).
$30e/$30f	784	iescpr	Vektor: User-Befehlswort listen ($51cd).
$310/$311	786	iescex	Vektor: Userbefehl ausführen ($4ba9).

Kernal-Vektoren:

$312/$313	786	itime	Nicht benutzt.
$314/$315	788	iirq	Interrupt-Vektor ($fa65).
$316/$317	790	ibrk	Break-Vektor ($b003).
$318/$319	792	inmi	Vektor für nicht maskierbare Interrupts ($fa40).
$31a/$31b	794	iopen	OPEN-Vektor ($efbd).
$31c/$31d	796	iclose	CLOSE-Vektor ($f188).
$31e/$31f	798	ichkin	CHKIN-Vektor ($f106).
$320/$321	800	ickout	CKOUT-Vektor ($f14c).
$322/$323	802	iclrch	CLRCH-Vektor ($f226).
$324/$325	804	ibasin	BASIN-Vektor ($ef06).

$326/$327	806	ibsout	BSOUT-Vektor ($ef79).
$328/$329	808	istop	STOP-Vektor ($f66e).
$32a/$32b	810	igetin	GETIN-Vektor ($eeeb).
$32c/$32d	812	iclall	CLALL-Vektor ($f222).
$32e/$32f	814	exmon	Monitor-Erweiterungsvektor ($b006).
$330/$331	816	iload	LOAD-Vektor ($f26c).
$332/$333	818	isave	SAVE-Vektor ($f54e).

Vektoren des Kernal-Editors:

$334/$335	820	iprctrl	Vektor: Zeichenausgabe mit Control ($c7b9).
$336/$337	822	iprshft	Vektor: Zeichenausgabe mit Shift ($c805).
$338/$339	824	ipresc	Vektor: Zeichenausgabe mit ESC ($c9c1).
$33a/$33b	826	ikeylog	Vektor: Auswertung der Tastaturabfrage ($c5e1).
$33c/$33d	828	ikeysto	Vektor: Tastencode abspeichern ($c6ad).

Zeiger auf Tastatur-Decodiertabellen:

$33e/$33f	830	keytab	Zeiger auf Decodiertabelle ohne zusätzliche Tasten ($fa80).
$340/$341	832	shftab	Decodiertabelle mit Shift ($fad9).
$342/$343	834	cbmtab	Decodiertabelle mit der CBM-Taste ($fb23).
$344/$345	836	ctrltab	Decodiertabelle mit Control ($fb8b).
$346/$347	838	alttab	Decodiertabelle mit ALT-Taste ($fa80).
$348/$349	840	dintab	Decodiertabelle für DIN-Tastenbelegung ($fbe4).

$34a-$353	842	keybuf	Tastaturpuffer.
$354-$35d	852	actmsk	10 Byte Bitmap der Tabulatorstopps (aktiver Bildschirmspeicher).
$35e-$361	862		4 Byte Bitmap der Zeilenverkettungen (aktiver Bereich).

$362-$36b	866	lfntab	Tabelle der logischen File-Nummern.
$36c-$375	876	gatab	Tabelle der Geräteadressen.
$376-$37f	886	satab	Tabelle der Sekundäradressen.

BASIC-Routinen in der Common Area:

$380-$39e	896	chrget	BASIC CHRGET-Routine.
$39f-$3aa	927	insubr0	Laden über PCRA und PCRC.
$3ab-$3b6	939	insubr1	Laden über PCRB und PCRD.
$3b7-$3bf	951	i24sr1	Laden über $24/$25 und PCRB + PCRD.

$3c0-$3c8	960	i26sr0	Laden über den Pointer $26/$27 bei Verwendung von PCRA und PCRC.
$3c9-$3d1	969	indtxt	Laden über den Pointer $3d/$3e unter Verwendung von PCRA und PCRC.

$3d2-$3d4	978	zero	Drei offenbar nicht benutzte Byte.
$3d5	981	pokbnk	Konfigurationsindex für POKE, PEEK, SYS etc.
$3d6-$3d9	982	tmpdes	Temporäre Zwischenspeicher für BASIC-Befehl INSTRING.
$3da	986	finbank	Konfigurationsindex, unter dem der String bei der String-/Fließkommawandlung zu finden ist. Es sei erwähnt, daß diese Speicherzelle bei den im Kapitel 5.3.4 angeführten Routinen keine Rolle spielt. Sie muß nicht vorbesetzt werden.
$3db-$3de	987	savsiz	Zwischenspeicher für SSHAPE und SPRSAV.
$3df	991	bits	Höchstwertiges Mantissen-Byte nach Verschiebung des FAC byteweise nach rechts. Normalerweise gleich 0, bei der Bildung eines 2er-Komplements gleich $ff. Diese Speicherzelle hat nur intern eine Bedeutung.
$3e0-$3e1	992	sprtmp	Sicherungsspeicher für den Programmzeiger beim SPRSAV-Befehl.
$3e2	994	fgbg	Grafikbildschirm: zusammengesetzter Farbcode. Oberes Halbbyte = Pixelfarbe, unteres Halbbyte = Hintergrund. Wird zum Beispiel benutzt, um den HiRes-Schirm zu löschen, indem das HiRes-Farb-RAM mit diesem Byte gefüllt wird.
$3e3	995	fgmcl	Multicolor-Grafik. Dasselbe Byte für die Multicolor-Grafik. Oberes Halbbyte = Pixelfarbe, unteres Halbbyte = Multicolorfarbe 1.
$3e4-$3ef	996		Freier Bereich.

$3f0-$3fc	1008	ramdisk	Hier liegt die Fortsetzung der Kernal-Routine DMA-CALL ($ff50), die später einmal eine angeschlossene RAM-Disk unterstützen soll. Von BASIC aus wird diese mit den drei Befehlen STASH, FETCH und SWAP ansprechbar sein. Bei normalem Ausbau des 128ers hat die vorliegende Routine keinerlei Bedeutung. Es läßt sich allerdings schon in Grundzügen erkennen, wie eine spätere RAM-Disk aufgebaut sein wird:

Der Controller der RAM-Disk wird im I/O-Bereich die Adressen $df01 bis $df08 (mindestens) belegen. Die einzelnen Register hätten die folgende Bedeutung:

$df01	Arbeitsmodus: $84 = STASH; $85 = FETCH ; $86 = SWAP
$df02/$df03	Adresse, woher Bytes zu holen sind. Der zugehörige Konfigurationsindex wäre der aus $3d5.
$df04/$df05	Adresse, wohin übertragen werden soll.
$df06	Konfigurationsindex dazu.
$df07/$df08	Anzahl der zu übertragenden Bytes.

$3fd-$3ff	1021		Freier Bereich.
$400-$7ff	1024	vicscn	VIC-Text-Video-RAM und Sprite-Zeiger.
$800-$9ff	2048	runstk	BASIC-Runtime-Stack. Hier werden die Einträge der FOR/NEXT-Schleifen, der GOSUB/RETURNs, der DO/LOOPs aufbewahrt.
$a00/$a01	2560	system	System-Restart-Vektor. Über diesen Vektor wird bei einem Reset gesprungen, um das laufende Anwenderprogramm, in der Regel BASIC, neu zu starten ($4003).
$a02	2562	dejavu	Flag: Kalt- oder Warmstart bei einem Reset durchführen. Ein Wert von $a5 in dejavu läßt einen Warmstart ausführen.
$a03	2563	palnts	Zeiger: PAL-System = $ff.
$a04	2564	initst	Flag für durchgeführte Initialisierungen. Flag für BASIC-Interruptroutinen freigegeben (Bit 0 = 1)!
$a05/$a06	2565	memstr	Untere Grenze des Systemspeichers in Bank 0.
$a07/$a08	2567	memsiz	Obere Grenze des Systemspeichers.

Die folgenden Speicherzellen für Band- und RS232-Betrieb möchte ich mir im Hinblick auf ihre untergeordnete Rolle sparen. Wer will, kann diese im Handbuch nachlesen. Interessanter wird es wieder bei:

$a1c	2588	serial	Temporäres Flag: schneller serieller Modus möglich.
$a1d-$a1f	2589	timer	Zwischenspeicher für die Systemuhr.
$a20	2592	xmax	Maximale Länge des Tastaturpuffers.
$a21	2593	pause	Flag für das Sperren der NO SCROLL-Taste.
$a22	2594	rptflg	Flag für Tastenwiederholung ($80 = alle Tasten, $40 = keine Taste, 0 = nur DEL, INST, Cursor-Tasten und Space.
$a23	2595	kount	Zähler für Geschwindigkeit der Tastenwiederholung.
$a24	2596	decay	Zähler für Verzögerung der Tastenwiederholung.
$a25	2597	sdelay	Wiederholungsverzögerung für Zeichensatzumschaltung.

$a26	2598	blnon	VIC-Cursor starr ($40), blinkend (0), starr und aus ($c0).
$a27	2599	blnsw	VIC-Cursor ganz ausschalten (ungleich 0).
$a28	2600	blncnt	Zähler für VIC-Cursor blinken.
$a29	2601	gdbln	Zeichen vor dem Blinken unter dem Cursor.
$a2a	2602	gdcol	Farbe des Zeichens unter dem Cursor.
$a2b	2603	curmod	VDC-Cursor-Modus (entspricht Register 10 des VDC.
$a2c	2604	vm1	Startadresse Video-RAM/Grafik Nummer 1 bei Verwendung von Splitscreens.
$a2d	2605	vm2	Nummer 2 bei Splitscreens. Startadresse der Bitmap.
$a2e	2606	vm3	Highbyte der Startadresse des VDC-Video-RAMs.
$a2f	2607	vm4	Highbyte der Startadresse des VDC-Attribut-RAMs.
$a30	2608	lintmp	Letzte Zeile für Eingabe vom Bildschirm.
$a31-$a33	2609		Verschiedene Zwischenspeicher.
$a34	2612	split	Rasterzeile für Splitscreens.
$a35	2613	curcol	Farbe unter dem Cursor beim VDC-Betrieb.
$a36-$a3f	2614		Verschiedene Zwischenspeicher.
$a40-$a59	2624	paspar	Passiver Bildschirmspeicher.
$a60-$a7f	2650	pasmsk	Passiver Bildschirmspeicher-Bereich Nummer zwei, Bitmaps.

Die hierauf folgenden Speicherzellen sind wieder mehr oder weniger uninteressant, weshalb sie weggelassen sind.

$b00-$bff	2816	tbuffr	Kassetten- und Bootpuffer.
$c00-$cff	3072	rs232i	RS232-Eingabepuffer.
$d00-$dff	3328	rs232o	RS232-Ausgabepuffer.
$e00-$fff	3584	sprites	Reservierter Bereich für Sprites.

$1000-$1009	4096	keylen	Länge der Funktionstasten-Strings.
$100a-$10ff	4106	keystr	Funktionstasten-Strings.
$1100-$1130	4352	dosstr	Bereich für DOS-Strings.

Hierauf folgen wieder eine Menge von einzelnen Speicherzellen der vielen BASIC-Routinen, die jedoch für den Maschinensprache-Programmierer keinen praktischen Nährwert haben.

$1131-$12ff	4401		Diverses.

$1300-$1bff	4864		Freier Bereich.
$1c00	7168		BASIC-Programmstart ohne Grafik.

Anhang A

Tastaturtabelle: Tastennummern, Tastencodes

Nummer	Taste	Tastencode	
0	DEL	$14	(20)
1	RETURN	$0d	(13)
2	Cursor rechts	$1d	(29)
3	F7	$88	(136)
4	F1	$85	(133)
5	F3	$86	(134)
6	F5	$87	(135)
7	Cursor nach unten	$11	(17)
8	Ziffer »3«	$33	(51)
9	w	$57	(87)
10	a	$41	(65)
11	4	$34	(52)
12	z	$5a	(90)
13	s	$53	(83)
14	e	$45	(69)
15	gleich 1 für Shift		1
16	5	$35	(53)
17	r	$52	(82)
18	d	$44	(68)
19	6	$36	(54)
20	c	$43	(67)
21	f	$46	(70)

Nummer	Taste	Tastencode	
22	t	$54	(84)
23	x	$58	(88)
24	7	$37	(55)
25	y	$59	(89)
26	g	$47	(71)
27	8	$38	(56)
28	b	$42	(66)
29	h	$48	(72)
30	u	$55	(85)
31	v	$56	(86)
32	9	$39	(57)
33	i	$49	(73)
34	j	$4a	(74)
35	0	$30	(48)
36	m	$4d	(77)
37	k	$4b	(75)
38	o	$4f	(79)
39	n	$4e	(78)
40	Plus	$2b	(43)
41	p	$50	(80)
42	l	$4c	(76)
43	Minus	$2d	(45)
44	Punkt	$2e	(46)
45	Doppelpunkt	$3a	(58)
46	Klammeraffe	$40	(46)
47	Komma	$2c	(44)
48	Pfundzeichen	$5c	(92)
49	Stern	$2a	(42)
50	Semikolon	$3b	(59)
51	HOME	$13	(19)
52	gleich 1 für Shift		1
53	Gleichheitszeichen	$3d	(61)
54	Pfeil nach oben	$5e	(94)
55	Schrägstrich	$2f	(47)
56	1	$31	(49)
57	Pfeil nach links	$5f	(95)
58	gleich 4 für Control		4
59	2	$32	(50)
60	Leertaste	$20	(32)

Nummer	Taste	Tastencode	
61	gleich 2 für CBM-Taste »C=«		2
62	q	$51	(81)
63	gleich 3 für STOP		3
64	HELP	$84	(132)
65	Zehnerblock 8	$38	(56)
66	Zehnerblock 5	$35	(53)
67	TAB	$09	(9)
68	Zehnerblock 2	$32	(50)
69	Zehnerblock 4	$34	(52)
70	Zehnerblock 7	$37	(55)
71	Zehnerblock 1	$31	(49)
72	ESC	$1b	(27)
73	Zehnerblock Plus	$2b	(43)
74	Zehnerblock Minus	$2d	(45)
75	LINE FEED	$0a	(10)
76	Zehnerblock ENTER	$0d	(13)
77	Zehnerblock 6	$36	(54)
78	Zehnerblock 9	$39	(57)
79	Zehnerblock 3	$33	(51)
80	gleich 8 für ALT		8
81	Zehnerblock 0	$30	(48)
82	Zehnerblock Punkt	$2e	(46)
83	Einzeltaste CRSR nach oben	$91	(145)
84	Einzeltaste CRSR nach unten	$11	(17)
85	Einzeltaste CRSR nach links	$9d	(157)
86	Einzeltaste CRSR nach rechts	$1d	(29)
87	nicht gültig		$ff
88	nicht gültig		$ff

Anhang B

Tabelle der BASIC-Befehle, Token und Adressen der BASIC-Befehle

Befehlswort	**1. Token**	**2. Token**	**Adresse**	
end	$80	–	$4bcd	(19405)
for	$81	–	$5df9	(24057)
next	$82	–	$57f4	(22516)
data	$83	–	$528f	(21135)
input #	$84	–	$5648	(22088)
input	$85	–	$5662	(22114)
dim	$86	–	$587b	(22651)
read	$87	–	$56a9	(22185)
let	$88	–	$53c6	(21446)
goto	$89	–	$59db	(23003)
run	$8a	–	$5a9b	(23195)
if	$8b	–	$52c5	(21189)
restore	$8c	–	$5aca	(23242)
gosub	$8d	–	$59cf	(22991)
return	$8e	–	$5262	(21090)
rem	$8f	–	$529d	(21149)
stop	$90	–	$4bcb	(19403)
on	$91	–	$53a3	(21411)
wait	$92	–	$6c2d	(27693)
load	$93	–	$912c	(37164)
save	$94	–	$9112	(37138)
verify	$95	–	$9129	(37161)

Befehlswort	1. Token	2. Token	Adresse	
def	$96	–	$84fa	(34042)
poke	$97	–	$80e5	(32997)
print #	$98	–	$553a	(21818)
print	$99	–	$555a	(21850)
cont	$9a	–	$5a60	(23136)
list	$9b	–	$50e3	(20706)
clr	$9c	–	$51f8	(20984)
cmd	$9d	–	$5540	(21824)
sys	$9e	–	$5885	(22661)
open	$9f	–	$918d	(38261)
close	$a0	–	$919a	(37274)
get	$a1	–	$5612	(22034)
new	$a2	–	$51d6	(20950)
else	$d5	–	$5391	(21393)
resume	$d6	–	$5f62	(24418)
trap	$d7	–	$5f4d	(24397)
tron	$d8	–	$58b4	(22708)
troff	$d9	–	$58b7	(22711)
sound	$da	–	$71ec	(29164)
vol	$db	–	$71c5	(29125)
auto	$dc	–	$5975	(22901)
pudef	$dd	–	$5f34	(24372)
graphic	$de	–	$6b5a	(27482)
paint	$df	–	$61a8	(25000)
char	$e0	–	$67d7	(26583)
box	$e1	–	$62b7	(25271)
circle	$e2	–	$668e	(26254)
gshape	$e3	–	$658d	(25997)
sshape	$e4	–	$642b	(25643)
draw	$e5	–	$6797	(26519)
locate	$e6	–	$6955	(26965)
color	$e7	–	$69e2	(27106)
scnclr	$e8	–	$6a79	(27257)
scale	$e9	–	$6960	(26976)
help	$ea	–	$5986	(22918)
do	$eb	–	$5fe0	(24544)
loop	$ec	–	$608a	(24714)
exit	$ed	–	$6039	(24633)
directory	$ef	–	$a07e	(41086)

Befehlswort	1. Token	2. Token	Adresse	
dsave	$f0	–	$a18c	(41356)
dload	$f1	–	$a1a7	(41383)
header	$f2	–	$a267	(41575)
scratch	$f3	–	$a2a1	(41633)
collect	$f4	–	$a32f	(41775)
copy	$f5	–	$a346	(41798)
rename	$f6	–	$a36e	(41838)
backup	$f7	–	$a37c	(41852)
delete	$f8	–	$5e87	(24199)
renumber	$f9	–	$5af8	(23288)
key	$fa	–	$610a	(24842)
monitor	$fb	–	$b000	(45056)

Token-Präfix $fe:

Befehlswort	1. Token	2. Token	Adresse	
bank	$fe	$02	$6bc9	(27593)
filter	$fe	$03	$7046	(28742)
play	$fe	$04	$6de1	(28129)
tempo	$fe	$05	$6fd7	(28631)
movspr	$fe	$06	$6cc6	(27846)
sprite	$fe	$07	$6c4f	(27727)
sprcolor	$fe	$08	$7190	(29072)
rreg	$fe	$09	$58bd	(22717)
envelope	$fe	$0a	$70c1	(28865)
sleep	$fe	$0b	$6db7	(27607)
catalog	$fe	$0c	$a07e	(41086)
dopen	$fe	$0d	$a11d	(41245)
append	$fe	$0e	$a134	(41268)
dclose	$fe	$0f	$a16f	(41327)
bsave	$fe	$10	$a1c8	(41416)
bload	$fe	$11	$a218	(41496)
record	$fe	$12	$a2d7	(41687)
concat	$fe	$13	$a362	(41826)
dverify	$fe	$14	$a1a4	(41380)
dclear	$fe	$15	$a322	(41762)
sprsav	$fe	$16	$76ec	(30444)
collision	$fe	$17	$7164	(29028)
begin	$fe	$18	$796c	(31084)

Befehlswort	1. Token	2. Token	Adresse	
bend	$fe	$19	$528f	(21135)
window	$fe	$1a	$72cc	(29388)
boot	$fe	$1b	$7335	(29493)
width	$fe	$1c	$71b6	(29110)
sprdef	$fe	$1d	$7372	(29554)
system	$fe	$1e	$4846	(18502)
stash	$fe	$1f	$aa1f	(43551)
fetch	$fe	$21	$aa24	(43556)
swap	$fe	$22	$aa29	(43561)
off	$fe	$23	$4846	(18502)
fast	$fe	$24	$77b3	(30643)
slow	$fe	$25	$77c4	(30660)

BASIC-Funktionen:

sgn	$b4	–	$8c65	(35941)
int	$b5	–	$8cfb	(36091)
abs	$b6	–	$8c84	(35972)
usr	$b7	–	$1218	(4632)
fre	$b8	–	$8000	(32768)
pos	$b9	–	$84d0	(34000)
sqr	$ba	–	$68a7	(36791)
rnd	$bb	–	$8434	(33844)
log	$bc	–	$89ca	(35274)
exp	$bd	–	$9033	(36915)
cos	$be	–	$9409	(37897)
sin	$bf	–	$9410	(37904)
tan	$c0	–	$9459	(37977)
atn	$c1	–	$94b3	(38067)
peek	$c2	–	$80c5	(32965)
len	$c3	–	$8668	(34408)
str$	$c4	–	$85ae	(34222)
val	$c5	–	$804a	(32842)
asc	$c6	–	$8677	(34423)
chr$	$c7	–	$85bf	(34239)
left$	$c8	–	$85d6	(34262)
right$	$c9	–	$860a	(34314)
mid$	$ca	–	$861c	(34332)

Befehlswort	1. Token	2. Token	Adresse	
go	$cb	–	$5a3d	(23101)
rgr	$cc	–	$8182	(33154)
rclr	$cd	–	$819b	(33179)
joy	$cf	–	$8203	(33283)
rdot	$d0	–	$9b0c	(39692)
dec	$d1	–	$8076	(32886)
hex$	$d2	–	$8142	(33090)
err$	$d3	–	$80f6	(33014)

BASIC-Funktionen mit Token-Präfix $ce:

pot	$ce	$02	$824d	(33357)
bump	$ce	$03	$837c	(33660)
pen	$ce	$04	$82ae	(33454)
rsppos	$ce	$05	$8397	(33687)
rsprite	$ce	$06	$831e	(33566)
rspcolor	$ce	$07	$8361	(33633)
xor	$ce	$08	$83e1	(33761)
rwindow	$ce	$09	$8407	(33799)
pointer	$ce	$0a	$82fa	(33530)

BASIC-Operationen:

Addition +	$aa	–	$8848	(34888)
Subtraktion -	$ab	–	$8831	(34865)
Multipl. *	$ac	–	$8a27	(35367)
Division /	$ad	–	$8b4c	(35660)
Potenz. ß	$ae	–	$8fc1	(36801)
and	$af	–	$4c89	(19593)
or	$b0	–	$4c86	(19590)
Vorzeichenwechsel	–	–	$8ffa	(36858)
not	$a8	–	$7930	(31024)
Vergleiche:			$4cb6	(19638)
Größer	$b1			
Gleich	$b2			
Kleiner	$b3			

Anhang C

6502 – BEFEHLSSATZ

Mnemonic	Adressierung	Code dezimal	Code sedezimal
ADC	adc # nn	105	69
	adc nn	101	65
	adc nn,x	117	75
	adc nnnn	109	6d
	adc nnnn,x	125	7d
	adc nnnn,y	121	79
	adc (nn,x)	97	61
	adc (nn),y	113	71
AND	and # nn	41	29
	and nn	37	25
	and nn,x	53	35
	and nnnn	45	2d
	and nnnn,x	61	3d
	and nnnn,y	57	39
	and (nn,x)	33	21
	and (nn),y	49	31
ASL	asl	10	0a
	asl nn	6	06
	asl nn,x	22	16
	asl nnnn	14	0e
	asl nnnn,x	30	1e
BCC	bcc nn	144	90

Mnemonic	Adressierung	Code dezimal	Code sedezimal
BCS	bcs nn	176	b0
BEQ	beq nn	240	f0
BIT	bit nn	36	24
	bit nnnn	44	2c
BMI	bmi nn	48	30
BNE	bne nn	208	d0
BPL	bpl nn	16	10
BVC	bvc nn	80	50
BVS	bvs nn	112	70
CLC	clc	24	18
CLD	cld	216	d8
CLI	cli	88	58
CLV	clv	184	b8
CMP	cmp # nn	201	c9
	cmp nn	197	c5
	cmp nn,x	213	d5
	cmp nnnn	205	cd
	cmp nnnn,x	221	dd
	cmp nnnn,y	217	d9
	cmp (nn,x)	193	c1
	cmp (nn),y	209	d1
CPX	cpx # nn	224	e0
	cpx nn	228	e4
	cpx nnnn	236	ec
CPY	cpy # nn	192	c0
	cpy nn	196	c4
	cpy nnnn	204	cc
DEC	dec nn	198	c6
	dec nn,x	214	d6
	dec nnnn	206	ce
	dec nnnn,x	222	de
DEX	dex	202	ca
DEY	dey	136	88
EOR	eor # nn	73	48
	eor nn	69	45
	eor nn,x	85	55
	eor nnnn	77	4d
	eor nnnn,x	93	5d
	eor nnnn,y	89	59

Mnemonic	Adressierung	Code dezimal	Code sedezimal
	eor (nn,x)	65	41
	eor (nn),y	81	51
INC	inc nn	230	e6
	inc nn,x	246	f6
	inc nnnn	238	ee
	inc nnnn,x	254	fe
INX	inx	232	e8
INY	iny	200	c8
JMP	jmp nnnn	76	4c
	jmp (nnnn)	108	6c
JSR	jsr nnnn	32	20
LDA	lda # nn	169	a9
	lda nn	165	a5
	lda nn,x	181	b5
	lda nnnn	173	ad
	lda nnnn,x	189	bd
	lda nnnn,y	185	b9
	lda (nn,x)	161	a1
	lda (nn),y	177	b1
LDX	ldx # nn	162	a2
	ldx nn	166	a6
	ldx nn,y	182	b6
	ldx nnnn	174	ae
	ldx nnnn,y	190	be
LDY	ldy # nn	160	a0
	ldy nn	164	a4
	ldy nn,x	180	b4
	ldy nnnn	172	ac
	ldy nnnn,x	188	bc
LSR	lsr	74	4a
	lsr nn	70	46
	lsr nn,x	86	56
	lsr nnnn	78	4e
	lsr nnnn	94	5e
NOP	nop	234	ea
ORA	ora # nn	9	09
	ora nn	5	05
	ora nn,x	21	15
	ora nnnn	13	0d

Mnemonic	Adressierung	Code dezimal	Code sedezimal
	ora nnnn,x	29	1d
	ora nnnn,y	25	19
	ora (nn,x)	1	01
	ora (nn),y	17	11
PHA	pha	72	48
PHP	php	8	08
PLA	pla	104	68
PLP	plp	40	28
ROL	rol	42	2a
	rol nn	38	26
	rol nn,x	54	36
	rol nnnn	46	2e
	rol nnnn,x	62	3e
ROR	ror	106	6a
	ror nn	102	66
	ror nn,x	118	76
	ror nnnn	110	6e
	ror nnnn,x	126	7e
RTI	rti	64	40
RTS	rts	96	60
SBC	sbc # nn	233	e9
	sbc nn	229	e5
	sbc nn,x	245	f5
	sbc nnnn	237	ed
	sbc nnnn,x	253	fd
	sbc nnnn,y	249	f9
	sbc (nn,x)	225	e1
	sbc (nn),y	241	f1
SEC	sec	56	38
SED	sed	248	f8
SEI	sei	120	78
STA	sta nn	133	85
	sta nn,x	149	95
	sta nnnn	141	8d
	sta nnnn,x	157	9d
	sta nnnn,y	152	99
	sta (nn,x)	129	81
	sta (nn),y	145	91
STX	stx nn	134	86

Mnemonic	Adressierung	Code dezimal	Code sedezimal
	stx nn,y	150	96
	stx nnnn	142	8e
STY	sty nn	132	84
	sty nn,x	148	94
	sty nnnn	140	8c
TAX	tax	170	aa
TAY	tay	168	ab
TSX	tsx	186	ba
TXA	txa	138	8a
TXS	txs	154	9a
TYA	tya	152	98

nn Zahl im Bereich von 0 bis $ff (0 bis 255)
nnnn Zahl zwischen 0 und $ffff (0 bis 65535)

Anhang D

ROUTINEN

Routinen der Common Area:

FETCH	$2a2	Zeichen über FETVEC = $2aa holen
STASH	$2af	Zeichen über STAVEC = $2b9 speichern
CMPARE	$2be	Zeichenvergleich über CMPVEC = $2c8
JSRFAR	$2cd	Weiter Unterprogrammaufruf aus Konfiguration JSFCNF = $2de
JMPFAR	$2e3	Weiter Sprung
CHRGET	$380	Nächstes Zeichen aus Programmtext lesen
CHRGOT	$386	Letztes Zeichen aus Text lesen
INSUBR0	$39f	Zeichen aus Bank 0 holen über $3a6
INSUBR1	$3ab	Zeichen aus Bank 1 holen über $3b2
I24SR1	$3b7	Zeichen in Bank 1 speichern über $24/$25
I26SR0	$3c0	Zeichen aus Bank 0 holen über $26/$27
INDTXT	$3c9	Zeichen aus Bank 0 holen über $3d/$3e

BASIC-Routinen:

SETVEC	$4251	BASIC-Vektoren restaurieren
SEARCH	$43e2	Key-Wort aus Tabelle suchen
LOOP	$4aa2	Interpreter-Hauptschleife
MISTAKE	$4d76	Fehlermeldung X ausgeben
WAIT	$4dc3	Eingabe-Warteschleife Direktmodus

GETLIN	$4f93	Eingabezeile nach $200 holen
LINSEA	$5064	Programmzeile suchen
LINNUM	$50a0	Zeilennummer lesen
LIST	$5123	Listen einer Zeile
USTLST	$516a	Befehlswort listen
NXTSTA	$528f	$3d/$3e auf nächstes Statement
OFFADD	$5292	Addition Y zu $3d/ $3e
OFFSET	$52a7	Offset-Trennzeichen X ermitteln
LETINT	$53e5	Zuweisung an Integer-Variable
LETREAL	$53fa	Zuweisung an REAL-Variable
LETSTR	$5405	Zuweisung an String-Variable
CROUT	$5598	Carriage Return ausgeben
STROUT	$55e2	String ((A,Y)) ausgeben
TONE	$726b	SID-Parameter übergeben
FRMNUM	$77d7	Numerischen Ausdruck holen
CHKNUM	$77da	Prüfung auf numerisch
CHKSTR	$77dc	Prüfung auf String
FRMEVL	$77ef	Auswertung beliebiger Ausdrücke
CHKKOM	$795c	Prüfung auf Komma
CHKSGN	$795e	Prüfung auf Zeichen im Akku
FETVAR	$7978	Variablenwert holen
GETVAR	$7aaf	Zeiger auf Variable setzen
INTOUT	$8e32	Ausgabe Integerzahl in (A,Y)
FRESTR	$877b	String holen (FRMEVL String)
STRPAR	$877e	String-Parameter holen
GETBYT	$87f1	Byte-Wert nach X holen
POKADR	$8803	Adresse, Komma, Byte-Wert holen
GETADR	$8812	Adresse nach (A,Y) holen
DECOUT	$8e32	Dezimalzahl (A,Y) ausgeben
DFTADR	$9c06	Adresse oder Default-Wert holen
DFTBYT	$9e1c	Byte-Wert oder Default-Wert holen
DOSPAR	$a3c3	Parameter für Disk-Befehle holen
COMCPL	$a667	Kommando-String zusammensetzen
FKTOINT	$af00	Umwandlung Fließkomma/Integer
INTTOFK	$af03	Umwandlung Integer/Fließkomma
FKTOSTR	$af06	Umwandlung Fließkomma/String
STRTOFK	$af09	Umwandlung String/Fließkomma
FCTOADR	$af0c	FAC nach Integer (auch negative Zahlen)
NORMINT	$af0f	Integer in normalisierte Fließkommazahl
SUBVAR	$af12	FAC := ((A,Y)) – FAC aus Bank 1
SUBARG	$af15	FAC := ARG – FAC

ADDVAR	$af18	FAC := ((A;Y)) + FAC aus Bank 1
ADDARG	$af1b	FAC := ARG + FAC
MULVAR	$af1e	FAC := ((A,Y)) * FAC aus Bank 1
MULARG	$af21	FAC := ARG * FAC
DIVVAR	$af24	FAC := ((A,Y)) / FAC aus Bank 1
DIVARG	$af27	FAC := ARG / FAC
LOGFAC	$af2a	FAC := ln (FAC)
INTFAC	$af2d	FAC := int (FAC)
SQRFAC	$af30	FAC := FAC ** 0.5
CHSSGN	$af33	FAC := – FAC
POTCON	$af36	FAC :=ARG ** ((A,Y)) aus Bank 0/ ROM
POTFAC	$af39	FAC := ARG ** FAC
EXPFAC	$af3c	FAC := exp (FEC)
COSFAC	$af3f	FAC := cos (FAC)
SINFAC	$af42	FAC := sin (FAC)
TANFAC	$af45	FAC := tan (FAC)
ATGFAC	$af48	FAC := arctan (FAC)
ROUNDF	$af4b	FAC runden
ABSFAC	$af4e	FAC := abs (FAC)
SGNFAC	$af51	Vorzeichen FAC ermitteln
CMPCON	$af54	Vergleich mit ((A,Y)) aus Bank 0/ ROM
RNDFAC	$af57	Zufallszahl erzeugen
MOVVARA	$af5a	ARG := ((A,Y)) aus Bank 1
MOVCONA	$af5d	ARG := ((A,Y)) aus Bank 0/ ROM
MOVVARF	$af60	FAC := ((A,Y)) aus Bank 1
MOVCONF	$af63	FAC := ((A,Y)) aus Bank 0/ ROM
STOFAC	$af66	((X,Y)) aus Bank 0 :=FAC
ARGTOF	$af69	FAC :=ARG
LINK	$af87	Zeilenlinker erzeugen
TOKEN	$af8a	Umwandlung in Token
FRMEVL	$af96	Auswertung beliebiger Ausdrücke (Sprungleiste)

Betriebssystemroutinen

PRINT	$c00c	Zeichen auf Bildschirm ausgeben
WINDOW	$c02d	Bildschirmfenster setzen
KEYSCN	$c55d	Tastaturmatrixabfrage
KEYLOG	$c5e1	Auswertung der Tastaturabfrage
CRSON	$cd6f	Cursor einschalten
CRSOFF	$cd9f	Cursor abschalten

FSTMOD	$ff47	CIA für fast mode setzen/ restaurieren
GACLOSE	$ff4a	Alle Files eines Gerätes schließen
C64MODE	$ff4d	Übergang in den 64er-Modus
DMACALL	$ff50	RAM-Disk ansprechen
BOOTCALL	$ff53	Diskette booten
PHOENIX	$ff56	Kaltstart
LKUPLA	$ff59	Logische File-Nummer suchen
LKUPSA	$ff5c	Sekundäradresse suchen
SWITCH	$ff5f	Umschaltung zwischen 40/80 Zeichen
DLCHR	$ff62	Zeichensatz in den VDC-Speicher bringen
PFKEY	$ff65	Funktionstaste mit String belegen
SETBNK	$ff68	Bänke für LSV und File-Namen setzen
GETCFG	$ff6b	Konfiguration zu Index X holen
PRIMM	$ff7d	Zeichenkette ausgeben
CINT	$ff81	Editor-Kaltstart
IOINIT	$ff84	Kaltstart I/O-Geräte
RAMTAS	$ff87	Zeropage löschen, Systemvektoren setzen
RESTOR	$ff8a	Kernal-Vektoren initialisieren
VEKTOR	$ff8d	Systemvektoren kopieren
SETMSG	$ff90	MSGFLG setzen
SECLST	$ff93	Sekundäradresse nach LISTEN setzen
SECTLK	$ff96	Sedundäradresse nach TALK senden
MEMTOP	$ff99	Obergrenze Systemspeicher setzen
MEMBOT	$ff9c	Untergrenze Systemspeicher setzen
KEY	$ff9f	Tastaturmatrixabfrage
SETTMO	$ffa2	IEEE-Timeout
IECIN	$ffa5	Zeichen vom IEC-Bus holen
IECOUT	$ffa8	Zeichen auf IEC-Bus ausgeben
UNTALK	$ffab	Untalk senden
UNLIST	$ffae	Unlisten senden
LISTEN	$ffb1	Listen senden
TALK	$ffb4	Talk senden
READST	$ffb7	Status lesen
SETPAR	$ffba	Logische File-Nummer, Gerät, Sek.-Adr. setzen
SETNAM	$ffbd	File-Namen-Parameter setzen
OPEN	$ffc0	File öffnen
CLOSE	$ffc3	File schließen
CHKIN	$ffc6	Eingabe auf File legen
CKOUT	$ffc9	Ausgabe auf File legen
CLRCH	$ffcc	Ein-/Ausgabe auf Default
BASIN	$ffcf	Zeichen vom akt. Eingabegerät holen

BSOUT	$ffd2	Zeichen auf akt. Ausgabegerät ausgeben
LOAD	$ffd5	File laden
SAVE	$ffd8	File abspeichern
SETTIM	$ffdb	Zeit setzen
RDTIM	$ffde	Zeit lesen
STOP	$ffe1	Stop abfragen
GETIN	$ffe4	Zeichen vom Eingabegerät holen
CLALL	$ffe7	Tabelle der Files löschen
UDTIM	$ffea	Uhrzeit erhöhen
SCRORG	$ffed	Bildschirmgröße holen
PLOT	$fff0	Cursor setzen/holen
IOBASE	$fff3	Adresse I/O-Bereich holen

Index